

**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FALCUTY OF ELECTRICAL AND ELECTRONIC ENGINEERING**

-----o0o-----



Final Capstone Project

**IMPLEMENT A CALCULATOR ON FPGA USING
RISC-V IF EXTENSION**

Lecturer: TS. Trần Hoàng Linh

Student: Nguyễn Phước Hải – 2111137

Dương Minh Đức - 2151009

HỒ CHÍ MINH CITY, April 2025



Số: _____/BKĐT
Khoa: **Điện – Điện tử**
Bộ Môn: **Điện Tử**

NHIỆM VỤ LUẬN VĂN TỐT NGHIỆP

- HỌ VÀ TÊN: Nguyễn Phước Hải MSSV: 2111137
Dương Minh Đức MSSV: 2151009
- NGÀNH: **ĐIỆN TỬ - VIỄN THÔNG** LỚP : DD21DV1, TT21HSA1
- Đề tài: Implement a calculator on FPGA using RISC-V IF extension
- Nhiệm vụ (Yêu cầu về nội dung và số liệu ban đầu):
 - Thiết kế máy tính bỏ túi trên FPGA áp dụng assembly sử dụng RISCV, nhập bằng keypad, hiển thị LCD 16x2.
 - Có thể thực hiện phép cộng, trừ, nhân, chia, sin, cos, tan.
 - Nhập và hiển thị kết quả số thực có phân thập phân.
- Ngày giao nhiệm vụ luận văn:
- Ngày hoàn thành nhiệm vụ:
- Họ và tên người hướng dẫn: TS. Trần Hoàng Linh
Nội dung và yêu cầu LVTN đã được thông qua Bộ Môn.

Tp.HCM, ngày..... tháng..... năm 20
CHỦ NHIỆM BỘ MÔN

NGƯỜI HƯỚNG DẪN CHÍNH

PHẦN DÀNH CHO KHOA, BỘ MÔN:

Người duyệt (chấm sơ bộ):.....
Đơn vị:.....
Ngày bảo vệ :
Điểm tổng kết:

ACKNOWLEDGEMENT

Dear Professor Tran Hoang Linh,

I would like to extend my heartfelt gratitude for your invaluable guidance and support throughout my internship project. Your insights and expertise have been instrumental in shaping my work and ensuring its successful progress.

Thank you for your patience, encouragement, and for always being available to provide feedback and direction. I am truly grateful for the opportunity to learn from you and for your commitment to my academic and professional development.

Looking forward to continuing this journey under your mentorship.

Sincerely,

Nguyen Phuoc Hai

Duong Minh Duc

Ho Chi Minh City, 25/04/2025

Student

CAPSTONE SUMMARY

In this capstone project, we implement a calculator on an FPGA using a custom RISC-V processor with the floating-point (F) extension. The primary goal is to extend a RISC-V core to support floating-point arithmetic, enabling the execution of complex mathematical operations such as addition, subtraction, multiplication and division.

To achieve this, we designed and integrated a Floating-Point Unit (FPU) within the processor pipeline, adhering to the IEEE-754 single-precision standard. The project also explores the RISC-V instruction set, particularly the F extension, and its implementation in hardware. By leveraging FPGA technology, we ensure efficient execution of floating-point operations in a reconfigurable environment.

The final application of our design is a functional calculator capable of handling arithmetic and scientific calculations. This project demonstrates the feasibility of extending RISC-V for floating-point computations on FPGA, with potential applications in embedded systems, scientific computing, and custom processor development.

Table Of Contents

1. Introduction Of The Project	1
1.1. Background and Motivation	1
1.2. Project Objective	1
1.3. Key Challenges.....	1
1.4. Reasons for choosing RISC-V.....	2
2. RISC-V Overview	3
2.1. Overview about RISC-V Processor.....	3
2.2. Introduction to Pipelining.....	3
2.3. The 5-Stage RISC-V Pipeline.....	4
2.4. Pipeline Hazards and Mitigation Strategies.....	5
3. Floating Point Arithmetic.....	7
3.1. Floating point number	7
3.2. Floating-point Representation	7
3.3. IEEE 754 Floating-Point Standard	8
3.4. Floating Point Conversions	11
3.4.1. Integer-To-Float Conversion	11
3.4.2. Float-to-Integer Conversion	12
3.4.3. Float to Int and Int to Float Implentmentation.....	13
3.5. Floating-point Adder and Subtractor.....	14
3.6. Floating-point Multiplier and Divider	17
3.6.1. Multiplier design	20
3.6.2. Divider Design	23
3.6.2.1. Newton Raphson theory	23
3.6.2.2. Floating point reciprocal	23
3.6.2.3. Newton Raphson Iteration	24
3.6.2.4. Derivation of initial values	24
3.6.2.5. Reciprocal unit	26

3.7.	Floating-point comparision.....	26
4.	RISC-V F Extension and Our Implementation Design	29
4.1.	The RISC-V ‘F’ Instruction Set.....	29
4.1.1.	Single-precision load and store instructions	30
4.1.2.	Single-precision floating-point computational instructions.....	30
4.1.3.	Single-precision floating-point move instructions	30
4.1.4.	Single-precision floating-point compare instructions	30
4.1.5.	Single-precision floating-point classify instruction	31
4.1.6.	Single-precision floating-point conversion instruction	31
4.1.7.	Single-precision floating-point sign injection instruction	32
4.1.8.	Single-precision floating-point Fused Multiply Addition instruction.....	32
4.2.	Overall System Design	33
4.3.	Detailed System Design.....	35
4.3.1.	Program Counter	35
4.3.2.	Register File	36
4.3.3.	Immediate Generator.....	37
4.3.4.	Branch Comparator.....	37
4.3.5.	Branch Control.....	38
4.3.6.	Arithmetic Logic Unit (ALU)	39
4.3.7.	Floating Point Unit (FPU).....	40
4.3.8.	ALU-FPU	41
4.3.9.	Control Unit.....	42
4.3.10.	Hazard unit.....	44
4.3.11.	Forwarding	46
4.3.12.	Branch Predict Table.....	48
5.	Calculator Application On FPGA Using RISC-V IF	51
5.1.	Overview	51

5.2.	About Keypad.....	53
5.3.	About LCD	58
5.4.	Flowchart For The Application	65
6.	Verification and results	67
6.1.	Verification for RISC-V IF	67
6.1.1.	Waveform testbench on Modelsim/Questasim	67
6.1.1.1.	Testing application on Questasim	67
6.1.1.2.	Testing Integer Instruction	69
6.1.1.3.	Testing application on Questasim	73
6.1.2.	Logic elements and Fmax	76
6.2.	Kit results.....	76
7.	Conclusion And Future Development	78
7.1.	Conclusion.....	78
7.2.	Future Development	78
8.	References.....	79

List of Figures

Figure 1. Block diagram of the RISC-V 5-stage pipelined processor with a 2-bit branch predictor. ...	6
Figure 2. Subranges and special values in floating-point number representations.	8
Figure 3. Single precision format.	10
Figure 4. Subnormals in the IEEE single-precision format.	10
Figure 5. Integer to Float conversion operation illustration.	12
Figure 6. Float to integer conversion illustration.	13
Figure 7. Our Float Convert Desgin.	13
Figure 8. Floating-point addition	14
Figure 9. Block diagram for floating point addition	15
Figure 10. Carry look ahead adder	17
Figure 11. Block diagram of a floating-point multiplier (divider).	18
Figure 12. Architecture of 3x3 Vedic Block	21
Figure 13. Architecture of 6x6 Vedic Block	21
Figure 14. Architecture of 12x12 Vedic Block	22
Figure 15. Architecture of 24x24 Vedic Block	22
Figure 16. Implementation of floating point reciprocal unit	23
Figure 17. Reciprocal Unit	26
Figure 18. Top Floating Point Comparision Module.	27
Figure 19. Floating Point Comparator.	27
Figure 20. Single precision floating-point load/store instruction.	30
Figure 21. Pipeline Diagram of the RISC-V Processor with 'F' Extension.	34
Figure 22. Program counter	35
Figure 23. Register File Block	36
Figure 24. Immediate Generator Block	37
Figure 25. Branch Comparator Block	37
Figure 26. Branch Control Block.	38

Figure 27. Operations ALU performs	39
Figure 28. ALU Block.....	39
Figure 29. FPU Block.....	40
Figure 30. Inside FPU Module.	41
Figure 31. ALU-FPU Block.....	42
Figure 32. Control Unit True Table.....	43
Figure 33. Control Unit Block	43
Figure 34. Hazard Unit Block.....	45
Figure 35. Forwarding Block.	46
Figure 36. Structure of Branch Predict Table Block.....	48
Figure 37. Predict bit diagram.....	49
Figure 38. Branch Predict Table Block.	49
Figure 39. Overview Calculator Flowchart	65
Figure 40. How we calculate the number using RISC-V	66
Figure 41. Waveform testbench on Questasim.....	67
Figure 42. Register Result For The Instruction.....	69
Figure 43. F-extension Testing on Modelsim.	73
Figure 44. Value Data For F-extension Testing on Modelsim.	73
Figure 45. First Number and Operator Input.....	76
Figure 46. Second Number Input.	77
Figure 47. Stack Result.	77

List of Tables

Table 1. IEEE 754 encoding of floating-point numbers.	9
Table 2. Instructions of 'F' extension.....	29
Table 3. Classification Mask of floating-point numbers	31
Table 4. Input/Output of Program Counter Block.....	35
Table 5. Inputs/Outputs of Register File	36
Table 6. Inputs/Outputs of Immediate Generator.....	37
Table 7. Inputs/Outputs of Branch Comparator	38
Table 8. Inputs/Outputs of Branch Control.....	39
Table 9. Inputs/Outputs of ALU	40
Table 10. Inputs/Outputs of FPU	41
Table 11. Inputs/Outputs of ALU-FPU	42
Table 12. Inputs/Outputs of Control Unit	44
Table 13. Inputs/Outputs of Hazard Unit.	46
Table 14. Inputs/Outputs of Forwarding.....	47
Table 15. Inputs/Outputs of Branch Predict Table.	50
Table 16. Comparison of Software and Hardware Approaches	58
Table 17. Overview of how ASCII characters are represented	64
Table 18. Comparison Pipelined I and F.....	76

1. Introduction Of The Project

1.1. Background and Motivation

As computing systems evolve, the demand for efficient arithmetic processing, particularly floating-point calculations, has grown significantly. While integer arithmetic is sufficient for many embedded and FPGA-based applications, floating-point operations are essential for scientific computing, engineering simulations, and advanced mathematical computations. Traditional FPGA-based processors often lack built-in floating-point support due to hardware complexity, leading to software-based approximations that introduce performance bottlenecks.

The RISC-V architecture, an open-source and modular instruction set architecture (ISA), provides a flexible solution by allowing extensions tailored to specific application needs. One such extension is the Floating-Point (F) extension, which introduces dedicated floating-point registers and operations, following the IEEE-754 standard. By implementing this extension in hardware, we can significantly improve computation speed and accuracy, making it a valuable enhancement for FPGA-based processors.

1.2. Project Objective

This project aims to design and implement a custom RISC-V processor with floating-point support on an FPGA, enabling efficient execution of floating-point arithmetic. The enhanced processor will be used to develop a scientific calculator capable of performing a wide range of mathematical operations, including:

- Basic arithmetic: Addition, subtraction, multiplication, and division.
- Advanced functions: Square root and trigonometric calculations (sin, cos, tan).

By integrating a Floating-Point Unit (FPU) within the RISC-V pipeline, we ensure seamless execution of these operations while maintaining high performance and accuracy. This project serves as a practical demonstration of how floating-point extensions can be incorporated into a custom processor design and deployed on FPGA hardware.

1.3. Key Challenges

To successfully implement floating-point support in our RISC-V processor, several challenges must be addressed:

- Floating-Point Unit (FPU) Design: Implementing a hardware-based FPU that follows the IEEE-754 standard while optimizing for FPGA resource constraints.
- Pipeline Integration: Modifying the RISC-V pipeline to efficiently handle floating-point operations while minimizing stalls and hazards.
- Calculator Implementation: Designing a user-friendly interface that utilizes the RISC-V core to evaluate complex mathematical expressions.

1.4. Reasons for choosing RISC-V

Implementing a calculator application on FPGA using a RISC-V core, instead of relying on dedicated hardware or IP blocks, offers notable flexibility and reusability. As an open and extensible instruction set architecture, RISC-V allows for the integration of custom instructions (e.g., for operations like \sqrt{x} , \sin , \cos , \tan), enabling easier adaptation of the system to evolving functional requirements.

Moreover, software-based implementation simplifies development through the use of high-level programming languages and facilitates debugging with standard tools (e.g., GDB). It also supports simulation and verification in software prior to FPGA deployment, thus accelerating the development cycle and enhancing system reliability.

In contrast, dedicated hardware typically provides better performance and lower resource usage for specific operations. However, it lacks flexibility and is less suitable for applications requiring frequent updates or extensions. Therefore, the choice of a RISC-V core is well-aligned with research and educational goals, where programmability, modularity, and experimentation are prioritized over raw computational performance.

2. RISC-V Overview

2.1. Overview about RISC-V Processor

RISC-V is an open standard Instruction Set Architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC). Originally developed at the University of California, Berkeley, RISC-V has gained widespread adoption in both academia and industry due to its simplicity, modularity, and extensibility. Unlike proprietary ISAs, RISC-V is freely available under open-source licenses, enabling researchers, developers, and hardware vendors to design, modify, and deploy custom processors without licensing fees.

The RISC-V architecture defines a small base integer instruction set (RV32I, RV64I, or RV128I) that can be extended with standard or custom instruction set extensions, such as those for multiplication and division (M), atomic operations (A), floating-point arithmetic (F and D), and compressed instructions (C). This modular design supports a wide range of computing domains, from embedded systems and IoT devices to high-performance computing and data centers.

One of the key advantages of RISC-V is its clean and orthogonal design, which simplifies hardware implementation and verification. Its open nature fosters innovation and collaboration, enabling the development of customizable processors tailored to specific application needs. Furthermore, a growing ecosystem of software tools, development platforms, and verification frameworks has accelerated the adoption and deployment of RISC-V processors across diverse computing environments.

2.2. Introduction to Pipelining

In modern processor design, pipelining is a fundamental technique used to improve instruction throughput by overlapping the execution of multiple instructions. Instead of waiting for one instruction to fully complete before starting the next, pipelining allows different parts of the processor to work on multiple instructions at the same time. This significantly enhances performance by increasing the number of instructions executed per clock cycle.

RISC-V, as a load-store architecture, efficiently supports pipelining by ensuring that most instructions have a uniform execution pattern. In our design, we implement the classic 5-stage RISC-V pipeline, which serves as the backbone for executing integer

instructions. This structured execution model is crucial for handling arithmetic operations, memory accesses, and control flow changes in an organized manner.

2.3. The 5-Stage RISC-V Pipeline

Each instruction progresses through the following five stages:

1. Instruction Fetch (IF)
 - The processor fetches the instruction from memory using the Program Counter (PC).
 - The PC is updated to point to the next instruction.
2. Instruction Decode (ID)
 - The instruction is decoded to determine its type (arithmetic, memory access, branch, etc.).
 - Source operands are read from the register file.
 - If necessary, immediate values are extracted.
3. Execute (EX)
 - The Arithmetic Logic Unit (ALU) performs computations such as addition, subtraction, and logical operations.
 - For branch instructions, the branch condition is evaluated.
 - For memory instructions, the memory address is calculated.
4. Memory Access (MEM)
 - Load instructions fetch data from memory, while store instructions write data to memory.
 - Instructions that do not involve memory skip this stage.
5. Write Back (WB)
 - The computed result (from ALU or memory) is written back to the register file if required.

By dividing execution into these five stages, the processor allows multiple instructions to be processed simultaneously, improving overall efficiency. However, this approach also introduces potential conflicts, known as pipeline hazards, which must be carefully managed.

2.4. Pipeline Hazards and Mitigation Strategies

Although pipelining increases efficiency, it also presents challenges where instructions depend on each other or compete for the same hardware resources. These issues, known as pipeline hazards, can cause incorrect execution or delays if not handled properly.

1. Data Hazards (Read-after-Write Conflicts)

A data hazard occurs when an instruction depends on the result of a previous instruction that has not yet completed. Solution:

- Forwarding (Bypassing): Instead of waiting for the write-back stage, the result can be forwarded directly from the EX or MEM stage.
- Stalling (Pipeline Bubble): If forwarding is not possible, a stall is inserted to delay execution.

2. Control Hazards (Branch Instructions)

Control hazards occur when a branch or jump instruction alters the execution flow, potentially causing incorrect instruction fetches. To minimize pipeline stalls, we implement a 2-bit branch predictor, which improves prediction accuracy compared to a simple static approach.

2-Bit Branch Prediction Mechanism:

Instead of assuming a branch is always taken or not taken, the 2-bit predictor tracks branch history and adjusts predictions dynamically.

Each branch is assigned a 2-bit saturating counter with four states:

- Strongly Taken
- Weakly Taken
- Weakly Not Taken
- Strongly Not Taken

If a branch is mispredicted, the counter moves one step toward the correct direction instead of immediately flipping, reducing misprediction penalties. This method provides better stability for branches that tend to be consistently taken or not taken. By incorporating 2-bit branch prediction, our design reduces the number of incorrect instruction fetches, minimizing pipeline stalls and improving overall efficiency.

3. Our RISC-V pipeline design

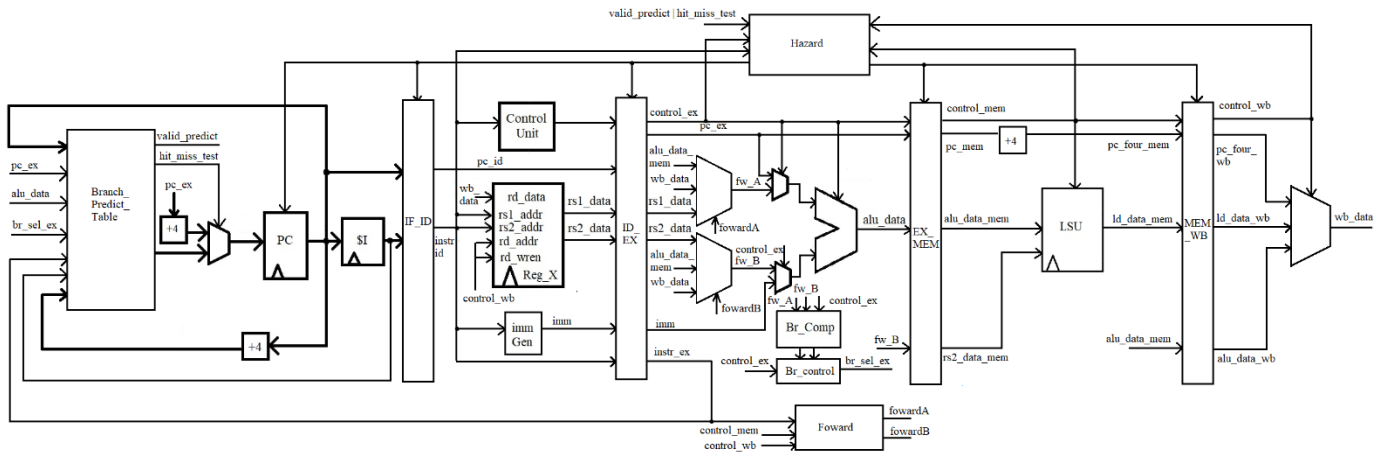


Figure 1. Block diagram of the RISC-V 5-stage pipelined processor with a 2-bit branch predictor.

The pipeline diagram visualizes the instruction and data flow, showing how the branch predictor, hazard detection, and forwarding logic work together. Specifically:

- Branch Predict Table is placed before the PC update, influencing instruction fetch decisions.
- Hazard Unit interacts with pipeline registers and control signals to detect and resolve hazards.
- Forwarding Unit is integrated in the EX stage to minimize stalls from data dependencies.
- Pipeline Registers (IF/ID, ID/EX, EX/MEM, MEM/WB) ensure smooth execution by passing data between stages.

3. Floating Point Arithmetic

3.1. Floating point number

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called *reals* in mathematics. Here are some examples of reals:

$3.14159265\dots_{\text{ten}}(\pi)$

$2.71828\dots_{\text{ten}}(e)$

0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in nanosecond)

$3,155,760,000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^9$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called scientific notation, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a normalized number, which is the usual way to write it. Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation: $1.0_{\text{two}} \times 2^{-1}$

To keep a binary number in the normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; binary point will do fine.

Computer arithmetic that supports such numbers is called floating point because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name *float* for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point.

A standard scientific notation for reals in the normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since real digits to the right of the binary point replace the unnecessary leading 0s.

3.2. Floating-point Representation

A designer of a floating-point representation must find a compromise between the size of the fraction (Notice that the fraction is also called the mantissa) and the size of the

exponent, because a fixed word size means you must take a bit from one to add a bit to the other. This tradeoff is between precision and range: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented.

Floating-point numbers are usually a multiple of the size of a word. The representation of a single precision floating-point number is shown below, where s is the sign of the floating-point number (1 meaning negative), $exponent$ is the value of the 8-bit exponent field (including the sign of the exponent), and $fraction$ is the 23-bit number.

These chosen sizes of exponent and fraction give a single precision arithmetic an extraordinary range. Fractions almost as small as $1.2_{\text{ten}} \times 10^{-38}$ and numbers almost as large as $3.4_{\text{ten}} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that overflow here means that the exponent is too large to be represented in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, we call this event underflow. This situation occurs when the negative exponent is too large to fit in the exponent field.

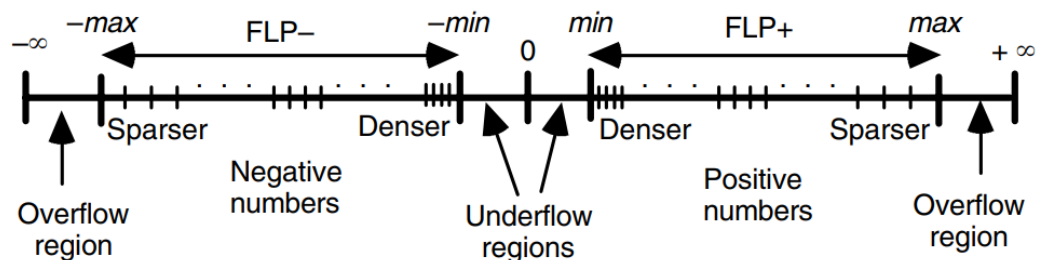


Figure 2. Subranges and special values in floating-point number representations.

3.3. IEEE 754 Floating-Point Standard

This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic. To pack even more bits into the number, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit. Hence, the number is

actually 24 bits long in single precision (implied 1 and a 23-bit fraction). To be precise, we use the term significand to represent the 24-bit number that is 1 plus the fraction and fraction when we mean the 23-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus $00 \dots 00_{\text{two}}$ represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^S \times (1.F) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field.

The table below shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing $+\infty$ or $-\infty$; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will output an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.) IEEE 754 even has a symbol for the result of invalid operations, such as $0/0$ or subtracting infinity from infinity. This symbol is *NaN*, for *Not a Number*. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when they are convenient.

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Table 1. IEEE 754 encoding of floating-point numbers.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{\text{two}} \times 2^{-1}$ would be represented in a single precision as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3. Single precision format.

The desirable notation must therefore represent the most negative exponent as $00 \dots 00_{\text{two}}$ and the most positive as $11 \dots 11_{\text{two}}$. This convention is called biased notation, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1 + 127_{\text{ten}}$, or $126_{\text{ten}} = 0111\ 1110_{\text{two}}$, and $+1$ is represented by $1 + 127$, or $128_{\text{ten}} = 1000\ 0000_{\text{two}}$. Biased exponent means that the value represented by a floating-point number really:

$$(-1)^S \times (1.F) \times 2^{(E-\text{bias})}$$

Subnormals, or subnormal values, are defined as numbers without a hidden 1 and with the smallest possible exponent. They are provided to make the effect of underflow less abrupt. In other words, certain small values that are not representable as normalized numbers, hence must be rounded to 0 if encountered in the course of computations, can be represented more precisely as subnormals. For example, $(0.0001)_{\text{two}} \times 2^{-126}$ is a subnormal that does not have a normalized representation in the IEEE single/short format. Because this “graceful underflow” provision can lead to cost and speed overhead in hardware, many implementations of the standard do not support subnormals, opting instead for the faster “flush to zero” mode.

Figure below shows the role of subnormals in providing representation points in the otherwise empty interval $(0, \text{min})$.

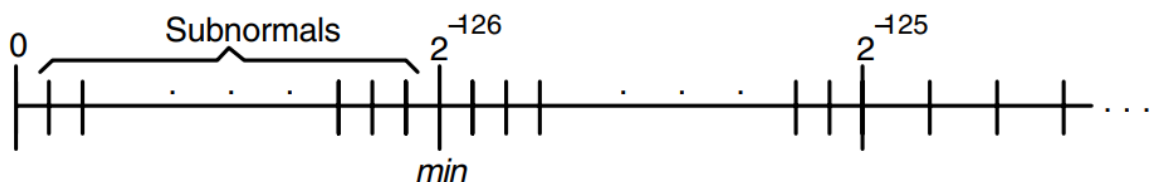


Figure 4. Subnormals in the IEEE single-precision format.

The IEEE 754-2008 standard also defines the four basic arithmetic operations (add, subtract, multiply, divide), as well as FMA and square-root, with regard to the expected precision in their results. Basically, the results of these operations must match the results

that would be obtained if all intermediate computations were carried out with infinite precision. Thus, it is up to the designers of floating-point hardware units adhering to IEEE 754-2008 to carry sufficient precision in intermediate results to satisfy this requirement. Finally, IEEE 754-2008 defines extended formats that allow implementations to carry higher precisions internally to reduce the effect of accumulated errors. Two extended binary formats are defined:

Single-extended: ≥ 11 bits for exponent, ≥ 32 bits for significand

(Bias unspecified, but exponent range must include $[-1022, 1023]$.)

The use of an extended format does not, in and of itself, guarantee that the precision requirements of floating-point operations will be satisfied. Rather, extended formats are useful for controlling error propagation in a sequence of arithmetic operations.

3.4. Floating Point Conversions

3.4.1. Integer-To-Float Conversion

First of all, since the MSB of the significand of normal numbers must always be 1 according to the IEEE 754 floating-point standard, the operand integer needs to be converted to this format. The steps of the algorithm are as follows.

- Step 1: Depending on whether the operand integer is interpreted as signed or unsigned, the absolute value of the number is taken and the 31 bits 0, which is the maximum shift number, is added to the end of the absolute value obtained.
- Step 2: Shift the temporary variable until the MSB of the significand become 1.
- Step 3: The exponent to be obtained will be equal to the shift amount. But this value is the actual value without bias added. For this reason, a bias of 127 is added to the final exponent for the single precision. Sign bit is equal 0 if the number is to be converted as unsigned, otherwise it is equal to input's original sign. For an illustration, the conversion of 1123412, an integer, to floatingpoint can be examined as follows in Figure 5.

In binary, 1123412 is represented as:

$$1123412_{10} = 00000000000100010010010001010100_2$$

In floating-point notation, it equals to:

$$(-1)^0 \times 2^{20} \times 1,0001001001000101010000_2$$

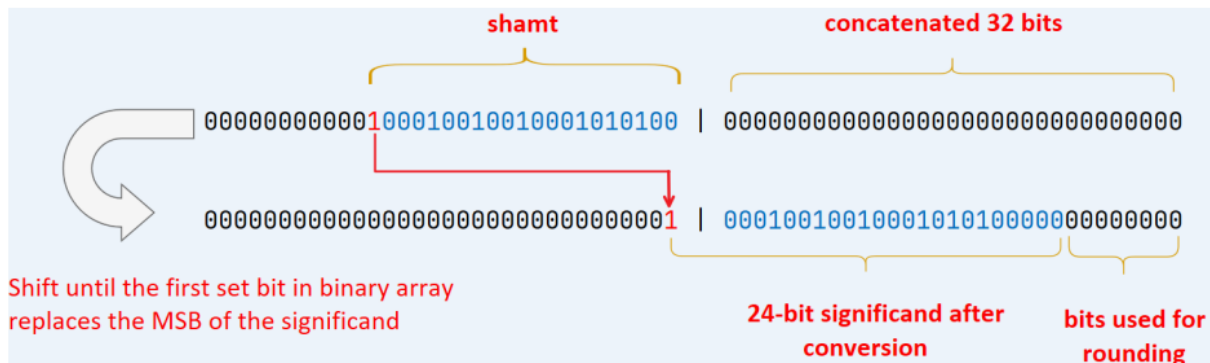


Figure 5. Integer to Float conversion operation illustration.

3.4.2. Float-to-Integer Conversion

In the integer conversion operation which works with a similar to the previous algorithm, if the exponent of operand floating point number is less than 0, the integer equivalent is directly equal to 0. Otherwise, the integer equivalent is determined when the significand is shifted until the exponent becomes 31. The reason for this is to shift the decimal point of the number by 31 digits. But, as mentioned before, when operand exponent is higher than 31, there are special cases where we cannot represent the operand floating point number with an integer. In these cases, output is determined by the directives in the RISC-V Manual. For an illustration, the conversion of 123423953.78845, a floating point number, to integer can be explained as follows. First of all, this number can be represented in single precision format as follows in equations and Figure 4.19:

$$123423953.78845_{10} = (-1)^0 \times 2^{26} \times 1,11010110110100110011010_2$$

After calculating the required shamt and performing the shift, the mantissa becomes:

$$(-1)^0 \times 2^{31} \times 0,0000111010110110100110011010_2$$

When we complete the remaining bits with 0, we get the following result.

$$00000111010110110100110011010000_2 = 123423952_{10}$$

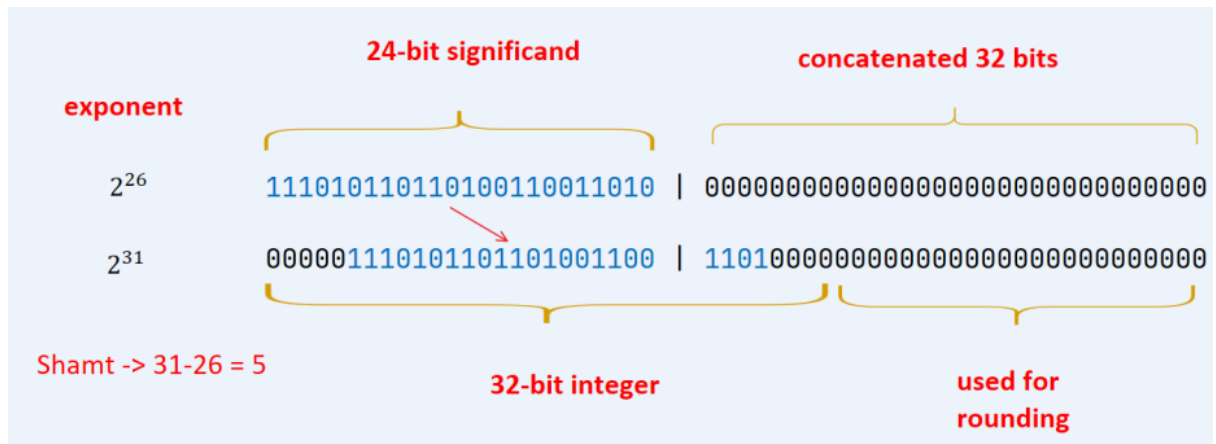


Figure 6. Float to integer conversion illustration.

3.4.3. Float to Int and Int to Float Implementation

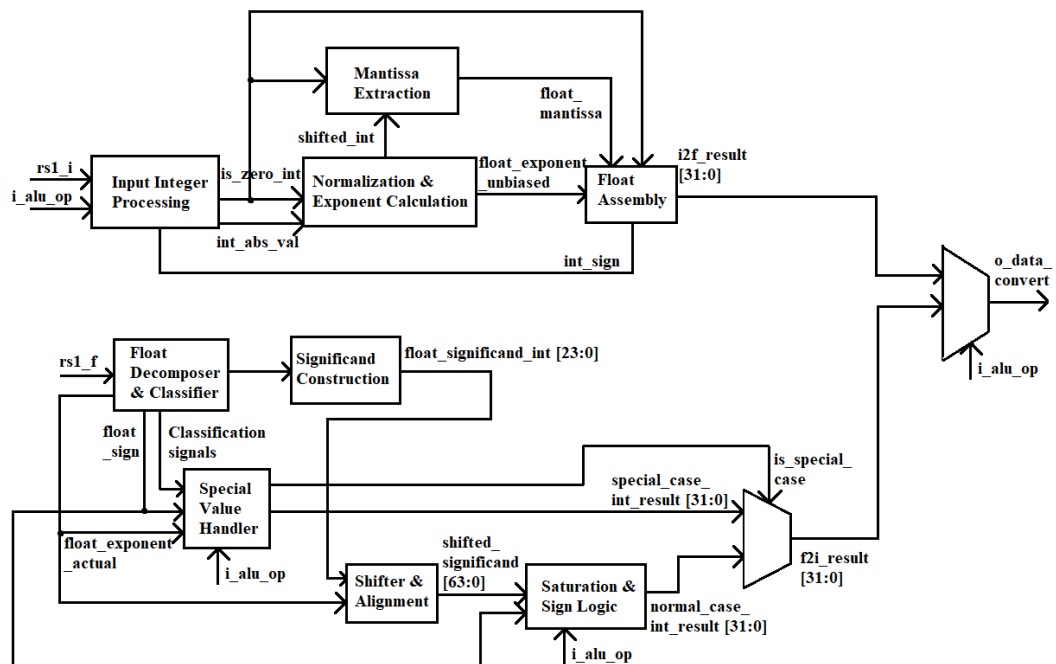


Figure 7. Our Float Convert Design.

3.5. Floating-point Adder and Subtractor

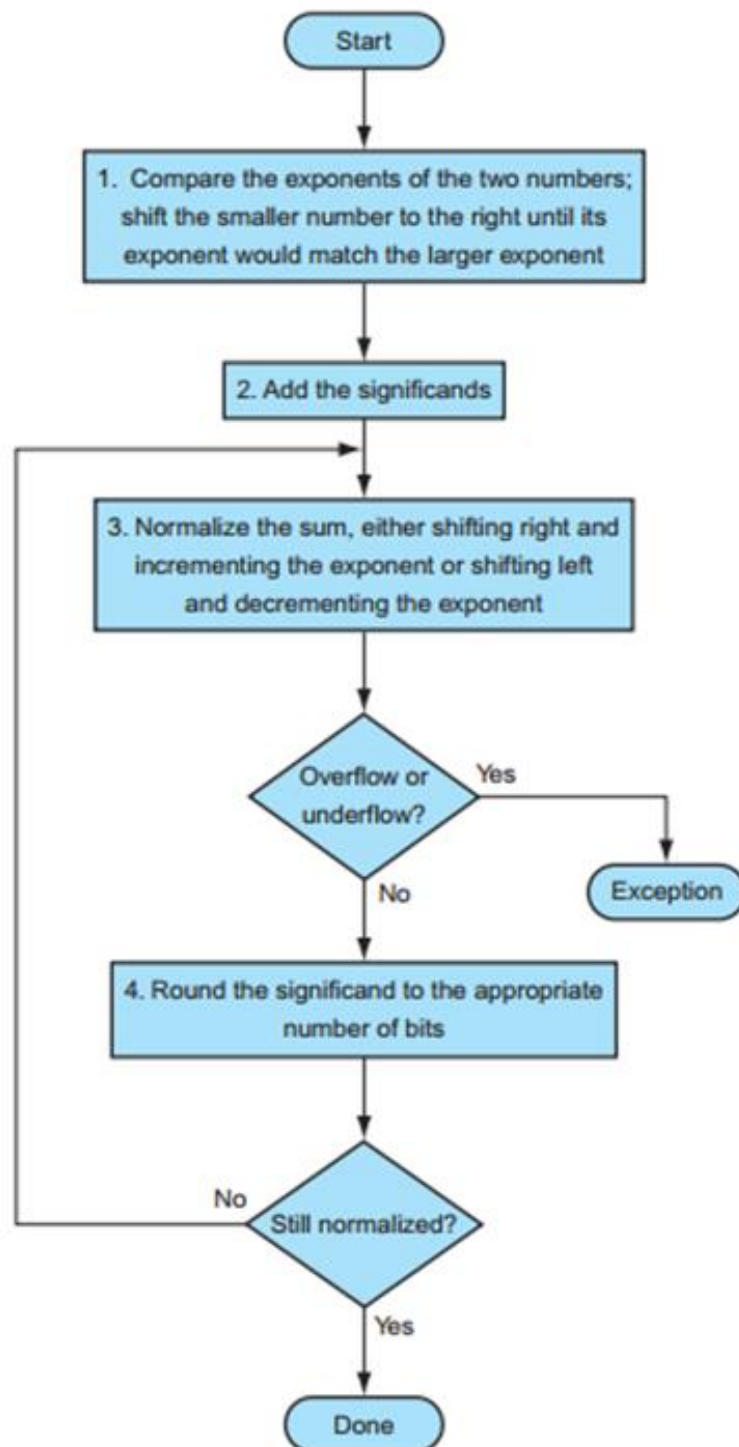


Figure 8. Floating-point addition

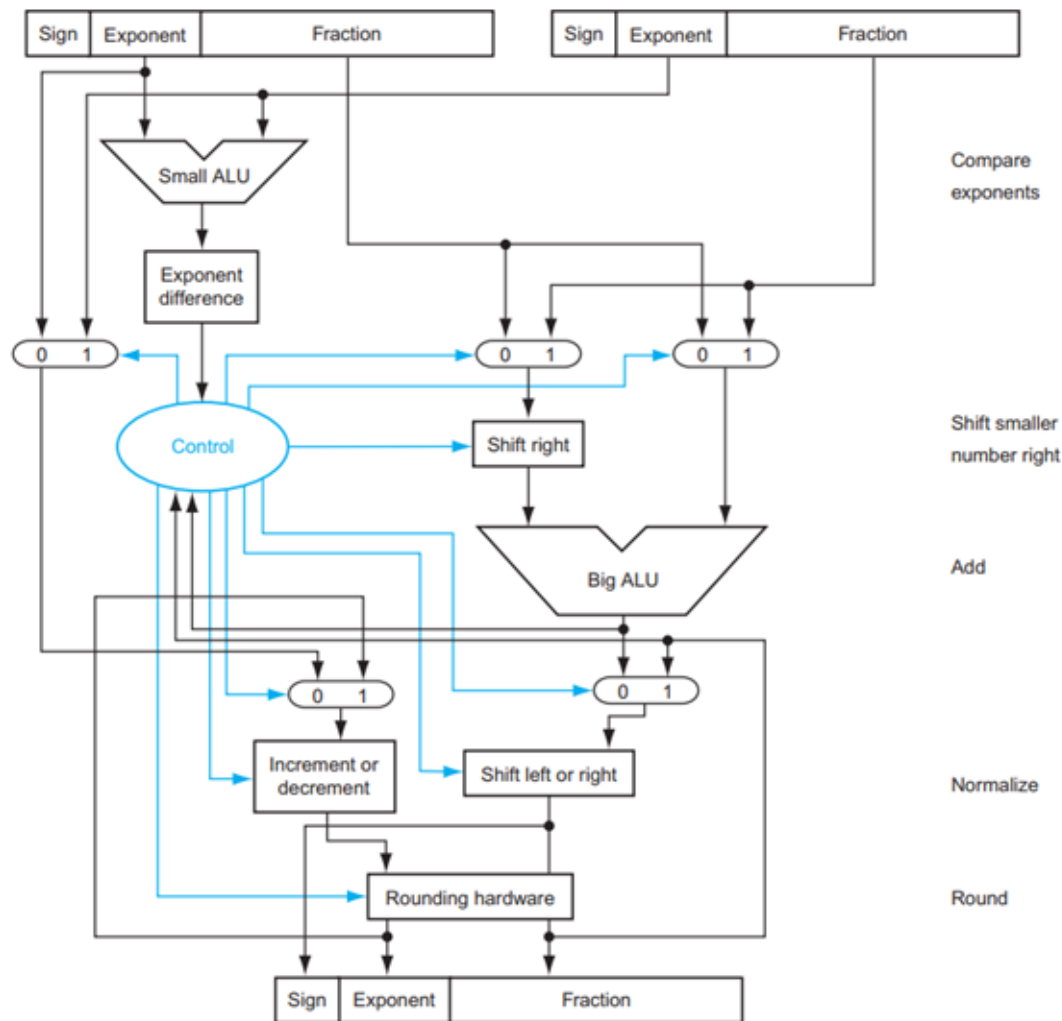


Figure 9. Block diagram for floating point addition

Floating-point addition/subtraction involves the following steps:

1. Align the Exponents:

- If the exponents of the two numbers differ, the smaller exponent is increased by shifting the mantissa of the smaller number.

2. Add/Sub the Mantissas:

- After aligning the exponents, the mantissas are added.
- If there is a carry from the addition, the result may need to be normalized.

3. Normalize the Result:

- If necessary, adjust the result to ensure the leading digit is non-zero.

4. Round the Result:

- The result may need to be rounded to fit within the available precision.

One common bottleneck in digital addition is the propagation delay caused by the carry from one bit to the next. The Carry Look-Ahead (CLA) mechanism is a solution that mitigates this delay, enabling faster arithmetic operations in CPUs and other digital systems. And we will use said CLA to calculate final addition phase.

3.5.1. Ripple Carry Adder

To understand the need for Carry Look-Ahead, it's essential to first grasp the functioning of a basic Ripple Carry Adder (RCA). In an RCA, each bit of the binary numbers being added produces a sum and a carry. The carry generated by each bit must be passed to the next higher bit to compute the final sum.

This sequential passing of carry bits results in a delay that increases linearly with the number of bits, known as the propagation delay. For large bit-widths (e.g., 32-bit or 64-bit numbers), this delay can be significant, slowing down the overall addition process.

The Carry Look-Ahead Adder (CLA) overcomes this problem by calculating the carry signals in advance, rather than waiting for the propagation from one bit to the next. The core idea is to predict whether a particular stage will generate or propagate a carry, allowing all carry bits to be determined in parallel, thus reducing the overall delay.

3.5.2. How Carry Look-Ahead Works

Carry Look-Ahead logic is based on two key concepts:

1. Generate (G) and Propagate (P) Functions:
 - Generate (G): This function indicates that a particular bit position will generate a carry regardless of the input carry. It is defined as $G_i = A_i \cdot B_i$, where A_i and B_i are the input bits.
 - Propagate (P): This function indicates that a carry-in will be propagated to the next higher bit. It is defined as $P_i = A_i \oplus B_i$.
2. Carry Look-Ahead Equations: The carry output for each bit can be expressed as:

$$C_{i+1} = G_i + (P_i + C_i)$$

Using this equation, the carry for each bit can be computed without waiting for the previous carry to propagate.

For example, in a 8-bit adder:

- $C_1 = G_0 + (P_0 \cdot C_0)$
- $C_2 = G_1 + (P_1 \cdot C_1) = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$
- And so on...

These equations can be implemented using combinational logic, allowing the carry bits to be computed in parallel.

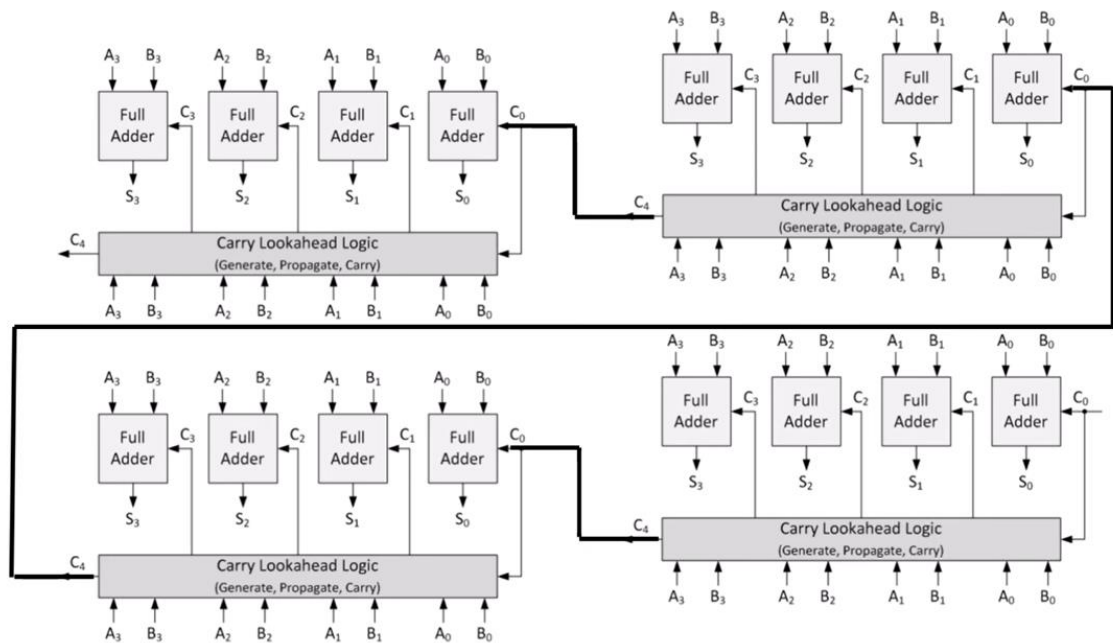


Figure 10. Carry look ahead adder

3.6. Floating-point Multiplier and Divider

A floating-point multiplier consists of a fixed-point multiplier for the significands, plus peripheral and support circuitry to deal with the exponents and special values (± 0 , $\pm \infty$, NaNs and subnormals). Figure 8 depicts a generic block diagram for a floating-point multiplier.

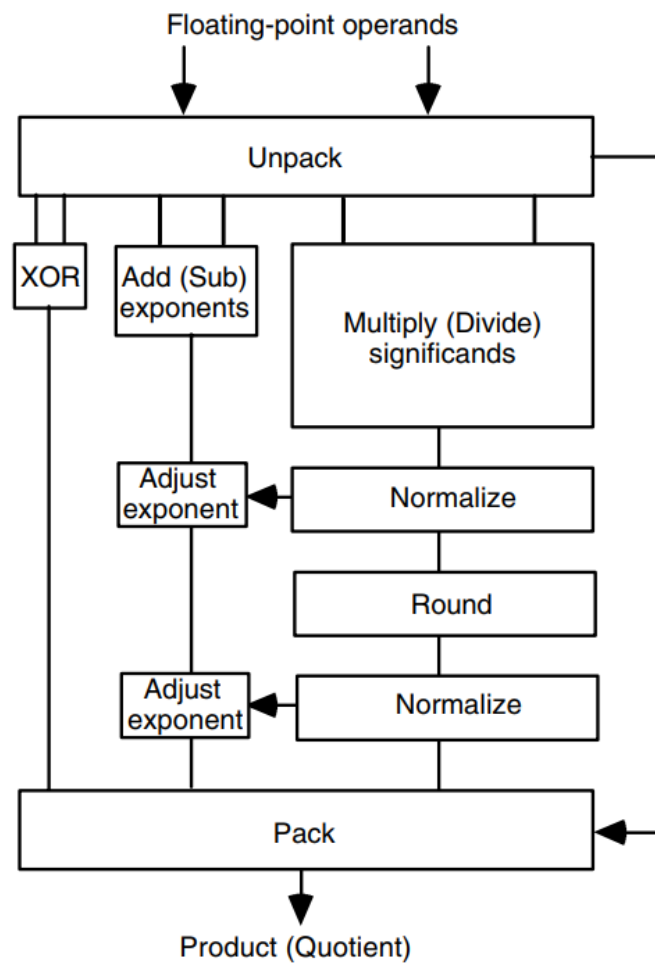


Figure 11. Block diagram of a floating-point multiplier (divider).

Unpacking involves:

- Separating the sign, exponent, and significand for each operand and reinstating the hidden 1.
- Converting the operands to the internal format, if different (e.g., single-extended or double-extended).
- Testing for special operands and exceptions (e.g., recognizing not-a-number inputs and bypassing the adder).

Packing the result involves:

- Combining the sign, exponent, and significand for the result and removing the hidden 1.
- Testing for special outcomes and exceptions (e.g., zero result, overflow, or underflow).

The sign of the product is obtained by XORing the signs of the two operands.

Tentative exponent is computed by adding the two biased exponents and subtracting the bias from the sum. With the IEEE 754-2008 short format, subtracting the bias of 127 can be easily accomplished by providing a carry-in of 1 into the exponent adder and subtracting 128 from the sum. This latter subtraction amounts to simply flipping the MSB of the result.

The significand multiplier is the slowest and most complex part of the unit. With the IEEE 754-2008 binary format, the product of the two unsigned significands, each in the range $[1, 2)$, will be in the range $[1, 4)$. Thus, the result may have to be normalized by shifting it one position to the right and incrementing the tentative exponent. Rounding the result may necessitate another normalizing shift and exponent adjustment. When each significand has a hidden 1 and l fractional bits, the significand multiplier is an unsigned $(l+1) \times (l+1)$ multiplier that would normally yield a $(2l+2)$ bit product. Since this full product must be rounded to $l+1$ bits at the output, it may be possible to discard the extra bits gradually as they are produced, rather than in a single step at the end.

A floating-point divider has the same overall structure as a floating-point multiplier as shown in figure 5. The two operands of floating-point division are unpacked, the resulting components pass through several computation steps, and the final result is packed into the appropriate format for output. Unpacking and packing have the same roles here as multiplier (the divide-by-0 exception is detected during unpacking). The sign of the quotient is obtained by XORing the operand signs.

A tentative exponent is computed by subtracting the divisor's biased exponent from the dividend's biased exponent and adding the bias to the difference. With the IEEE 754-2008 short format, the bias of 127 must be added to the difference of the two exponents. Since adding 128 is simpler than adding 127, we can compute the difference less one by holding c_{in} to 0 in a 2's-complement subtraction (normally, in 2's-complement subtraction, $c_{in} = 1$) and then flipping the MSB of the result. The significand divider is the slowest and the most complex part of the unit shown in Fig. 18.6. With the IEEE 754-2008 format, the ratio of two significands in $[1, 2)$ is in the range $(1/2, 2)$. Thus, the result may have to be normalized by shifting it one position to the left and decrementing the

tentative exponent. Rounding the result may necessitate another normalizing shift and exponent adjustment.

As was the case for fixed-point multipliers and dividers, floating-point multipliers and dividers can share much hardware. In particular, when the significand division is performed by one of the convergence methods, little additional hardware is required to convert a floating-point multiplier into a floating-point multiply/divide unit.

3.6.1. Multiplier design

The performance of Mantissa calculation Unit dominates overall performance of the Floating Point Multiplier. This unit requires unsigned multiplier for multiplication of 24x24 BITS. The Vedic Multiplication technique is chosen for the implementation of this unit. This technique gives promising result in terms of speed and power. The Vedic multiplication system is based on 16 Vedic sutras or aphorisms, which describes natural ways of solving a whole range of mathematical problems. Out of these 16 Vedic Sutras the Urdhva-triayakbhyam sutra is suitable for this purpose. In this method the partial products are generated simultaneously which itself reduces delay and makes this method fast. In this proposed multiplier the base block used as first stage implementation is 3x3 block which is shown in figure 6.

The 3x3 block consists of two half adders, one full adder and three 2 bit adders as shown in figure 6. From this 3x3 block, 6x6 multiplier block is designed. From this 6x6 multiplier block, 12x12 multiplier block is designed and similarly from this 12x12 multiplier block, 24x24 multiplier block is designed and implemented. These blocks require Vedic multipliers and ripple carry adders for getting the final output.

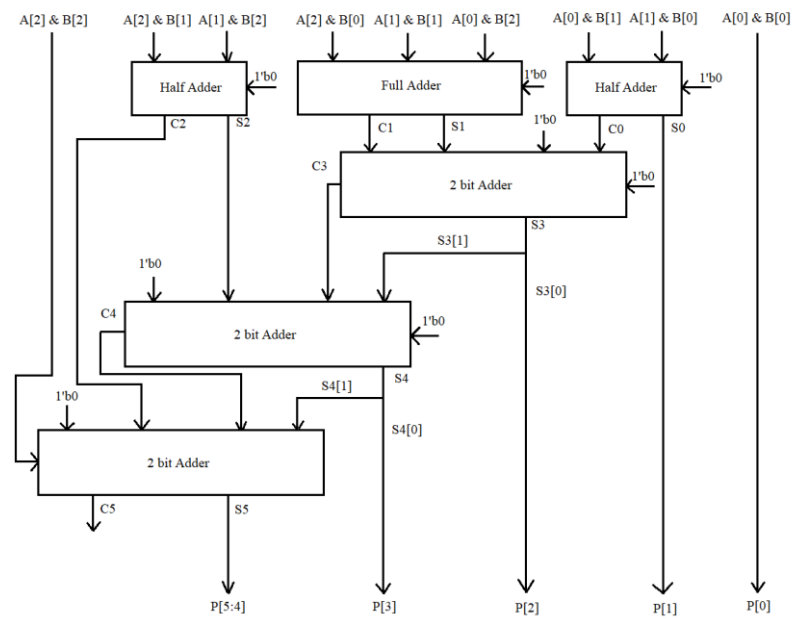


Figure 12. Architecture of 3x3 Vedic Block

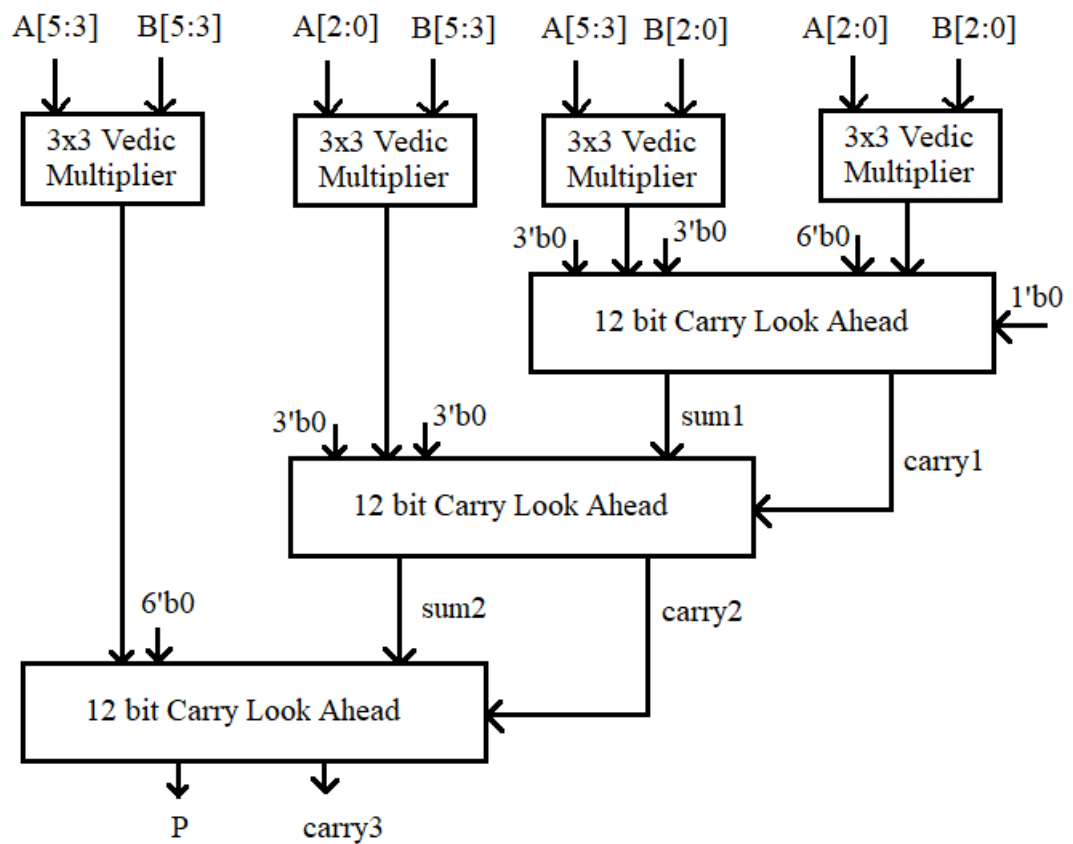


Figure 13. Architecture of 6x6 Vedic Block

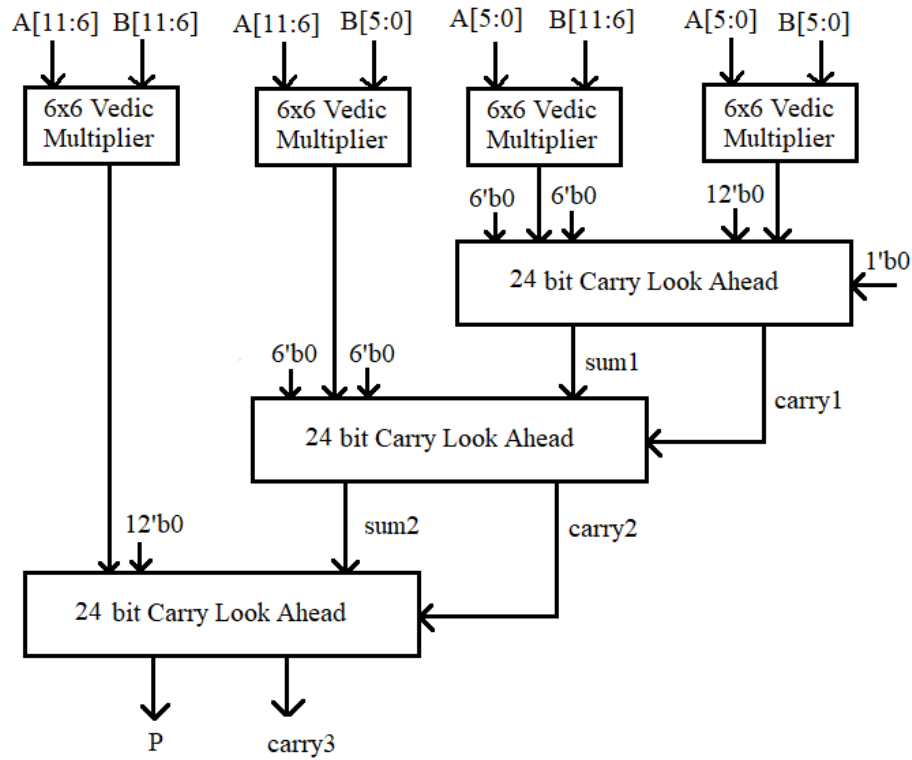


Figure 14. Architecture of 12x12 Vedic Block

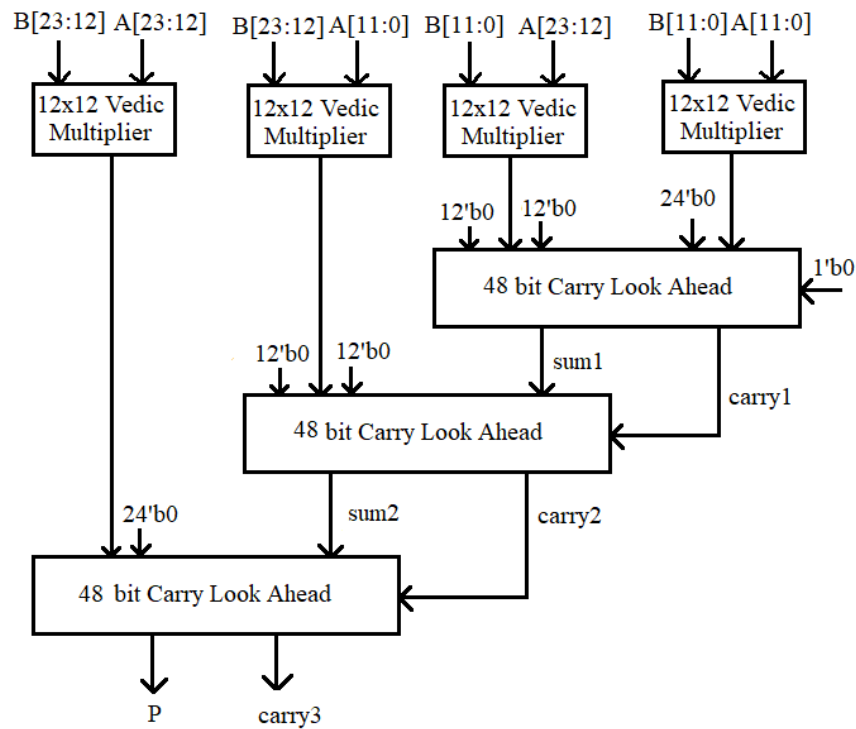


Figure 15. Architecture of 24x24 Vedic Block

3.6.2. Divider Design

3.6.2.1. Newton Raphson theory

Newton–Raphson uses Newton's method to find the reciprocal of D and multiply that reciprocal by N to find the final quotient Q

The steps of Newton–Raphson division are:

- Calculate an estimate X_0 for the reciprocal $1/D$ of the divisor D
- Compute successively more accurate estimates x_1, x_2, x_3 of the reciprocal. This is where one employs the Newton–Raphson method as such.
- Compute the quotient by multiplying the dividend by the reciprocal of the divisor: $Q = NS$

3.6.2.2. Floating point reciprocal

The sign, exponent, and mantissa of an IEEE-754 floating number X are represented as S_x , E_x , and M_x respectively. The reciprocal ($R=1/X$) of this number can be computed using $S_r = S_x$, $E_r = -E_x$, $M_r = 1/M_x$.

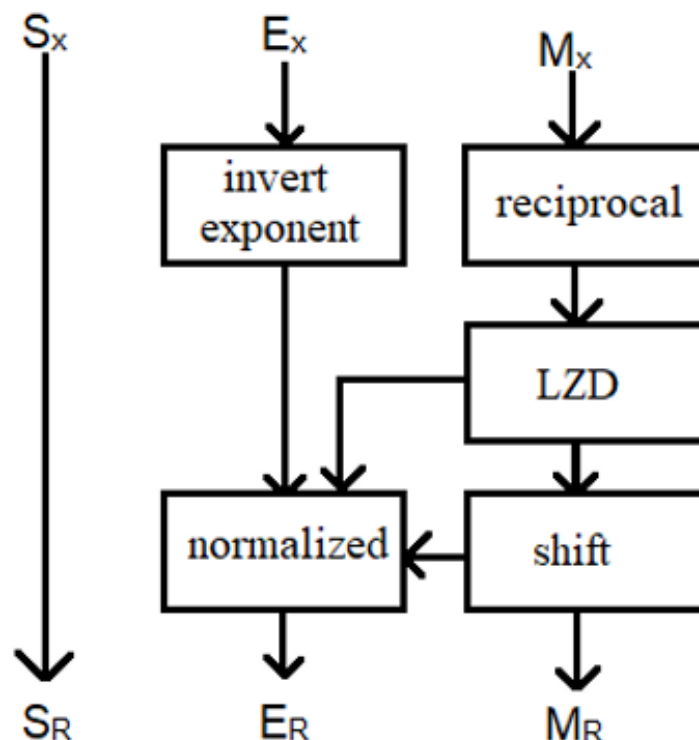


Figure 16. Implementation of floating point reciprocal unit

3.6.2.3. Newton Raphson Iteration

The Newton-Raphson algorithm is widely used in solving non-linear equations. The Newton-Raphson technique needs an initial value x_0 , which is referred as initial guess for the root. The derivation is carried out by Taylor series. The function $f(x)$ can be written using Taylor series expansion in period $x-x_0$ as:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 = 0$$

The series can be shortened by throwing the second term and obtain the Newton-Raphson iteration formula as:

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

A more general form of equation can be written as:

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

$$x_{i+1} = x_i(2 - M_{x_i})$$

An initial look-up table is used to obtain an approximate value of the root. Each iteration doubles the accuracy of the result.

3.6.2.4. Derivation of initial values

The n-bit mantissa M is represented as:

$$1.m_1m_2m_3 \dots m_{n-1} \quad (m_i \in \{0,1\}, i = 1 \dots n)$$

When M is divided into two parts M1 and M2 as:

$$M_1 = 1.m_1m_2m_3 \dots m_m$$

and

$$M_2 = 0.m_{m+1}m_{m+2}m_{m+3} \dots m_{n-1}$$

The first-order Taylor expansion of M^p of number M is between M_1 and $M_1 + 2^{-(m-1)}$ and is expressed as:

$$(M_1 - 2^{-(m-1)})^{p-1} \cdot (M_1 + 2^{-(m-1)} + p \cdot (M_2 - 2^{-(m-1)}))$$

The equation can be expressed as:

$$C \cdot M$$

where $C = (M_1 - 2^{-(m-1)})^{p-1}$ and

$$M = M_1 + 2^{-(m-1)} + p \cdot (M_2 - 2^{-(m-1)})$$

C can be read from a lookup-table which is addressed by M_1 , without leading one. The lookup table contains the 2^m of C values of M for special values of p , where it is -2^0 for reciprocal of M . The size of the required ROM for the lookup table is about $2^m \times 2m$ bits.

The initial approximation of floating point number $M^{(-1)}$ is computed by multiplication of term C with modified operand M . The modified form of M is obtained by only complementing M_2 bitwise. The last term can be ignorable.

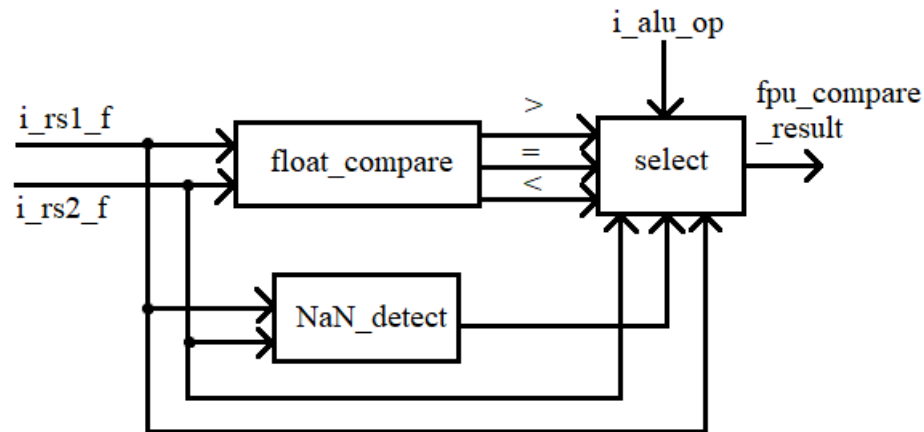


Figure 18. Top Floating Point Comparison Module.

i_rs1_f and i_rs2_f will be inputted compare block to find out which will be greater than, equal or less than. These two input also inputted in NaN detect to know which one of them is NaN value (this will affect the output of the comparison of flt, fle and feq). The select block will input compare flag and NaN flag along with 5 selector (which is 5 float instructions flag).

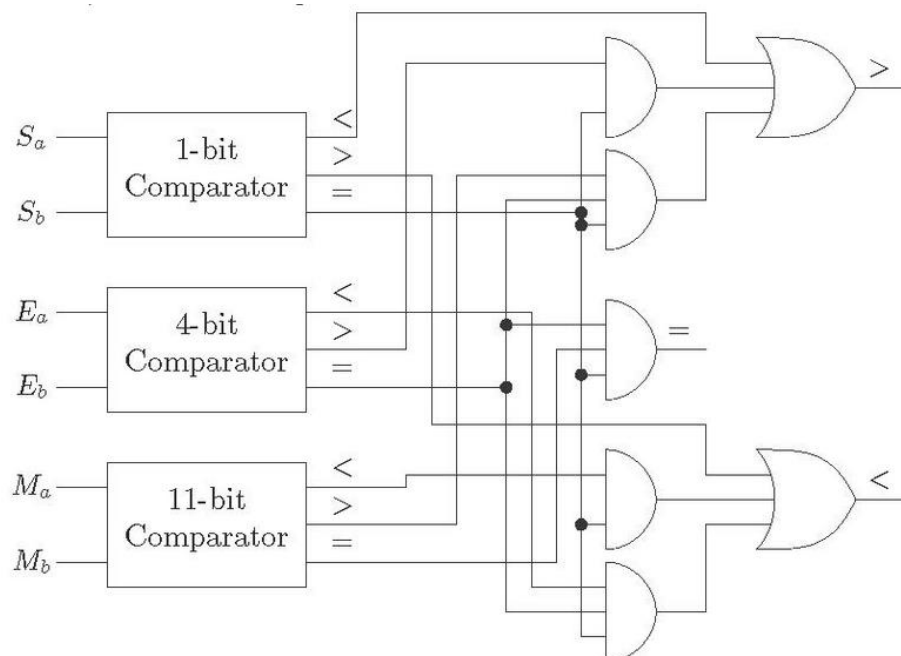


Figure 19. Floating Point Comparator.

The steps involving the comparison of two floating point numbers rs1 and rs2 are:

- Compare the sign bits. If both are of same sign then proceed to the next step.
- Compare the exponents. If both exponents are also equal then go to the next step.
- Compare the mantissas. Generate three outputs flags.

Here we have shown a comparator for two 32-bit floating point numbers $rs1$ and $rs2$. This 32-bit comparator is realized using 3 comparators, 1-bit comparator (Sign check), 8-bit comparator (Exponent check) and 23-bit comparator (mantissa check). Two sign bits are compared using a simple 1-bit comparator. Here we will use the opposite of the convention used in case of 1 bit comparator. This is because, here if the sign bit is '1' then the number is negative and thus lesser. If sign of both the numbers are same then the exponents are compared. Here a 8-bit comparator is used to compare the exponents. If both the numbers have same sign and their exponents are also equal then at the last step mantissas are compared. The two mantissas are compared using a 23-bit comparator.

4. RISC-V F Extension and Our Implementation Design

4.1. The RISC-V 'F' Instruction Set

Single-precision floating-point computational instructions of RISC-V, designated as "F" and conforming to the IEEE 754-2008 arithmetic standard, are briefly elaborated in Table 2.

Instructions	Name	Description (C)
flw	Flt Load Word	$f[rd] = M[x[rs1] + sext(offset)][31:0]$
fsw	Flt Store Word	$M[x[rs1] + sext(offset)] = f[rs2][31:0]$
fmadd.s	Flt fused Multiply Addition	$f[rd] = f[rs1] \times f[rs2] + f[rs3]$
fmsub.s	Flt fused Multiply Subtraction	$f[rd] = f[rs1] \times f[rs2] - f[rs3]$
fnmsub.s	Flt Neg Fused Multiply Addition	$f[rd] = -f[rs1] \times f[rs2] + f[rs3]$
fnmadd.s	Flt Neg Fused Multiply Subtraction	$f[rd] = -f[rs1] \times f[rs2] - f[rs3]$
fadd.s	Flt Add	$f[rd] = f[rs1] + f[rs2]$
fsub.s	Flt Sub	$f[rd] = f[rs1] - f[rs2]$
fmul.s	Flt Mul	$f[rd] = f[rs1] \times f[rs2]$
fdiv.s	Flt Div	$f[rd] = f[rs1] / f[rs2]$
fmin.s	Flt Minimum	$f[rd] = \min(f[rs1], f[rs2])$
fmax.s	Flt Maximum	$f[rd] = \max(f[rs1], f[rs2])$
fmv.x.w	Move Float to Int	$x[rd] = sext(f[rs1][31:0])$
fmv.w.x	Move Int to Float	$f[rd] = x[rs1][31:0]$
fcvt.w.s	Covert Float to Int	$x[rd] = sext(s32_{\{f32\}}(f[rs1]))$
fcvt.wu.s	Covert Float to Unsigned Int	$x[rd] = sext(u32_{\{f32\}}(f[rs1]))$
fcvt.s.w	Covert Signed Int to Float	$f[rd] = f32_{\{s32\}}(x[rs1])$
fcvt.s.wu	Covert Unsigned Int to Float	$f[rd] = f32_{\{u32\}}(x[rs1])$
fsgnj.s	Flt Sign Injection	$f[rd] = \{f[rs2][31], f[rs1][30:0]\}$
fsgnjn.s	Flt Sign Injection Neg	$f[rd] = \{\sim f[rs2][31], f[rs1][30:0]\}$
fsgnjx.s	Flt Sign Injection Xor	$f[rd] = \{f[rs1][31] \wedge f[rs2][31], f[rs1][30:0]\}$
feq.s	Float Equality	$x[rd] = (f[rs1] == f[rs2]) ? 1 : 0$
flt.s	Float Less Than	$x[rd] = (f[rs1] < f[rs2]) ? 1 : 0$
fle.s	Float Less / Equal	$x[rd] = (f[rs1] <= f[rs2]) ? 1 : 0$
fclass.s	Float Classify	$x[rd] = \text{classifys}(f[rs1])$

Table 2. Instructions of 'F' extension

There are several considerations when adding an FPU to the processor that will add considerable complexity. First, we will need to implement changes to the Control Unit to correctly interpret this extended set of instructions and store information about whether operations are integer or floating-point. The RISC-V floating-point extension uses 32 additional 32-bit registers for floating-point operations, so we'll need to include a second register file for these floating-point registers.

4.1.1. Single-precision load and store instructions

The load and store instructions, named as FLW and FSW can be seen at Figure 1, loads a single-precision floating-point value to floating-point register from memory and stores a single precision value of floating-point register to memory, respectively. Load and store instructions use a base address in source register and 12-bit signed byte offset as base+offset addressing method as integer base ISA does. In the implementation of these commands, it is not much different from normal load and store as only the correct register bank should be selected.

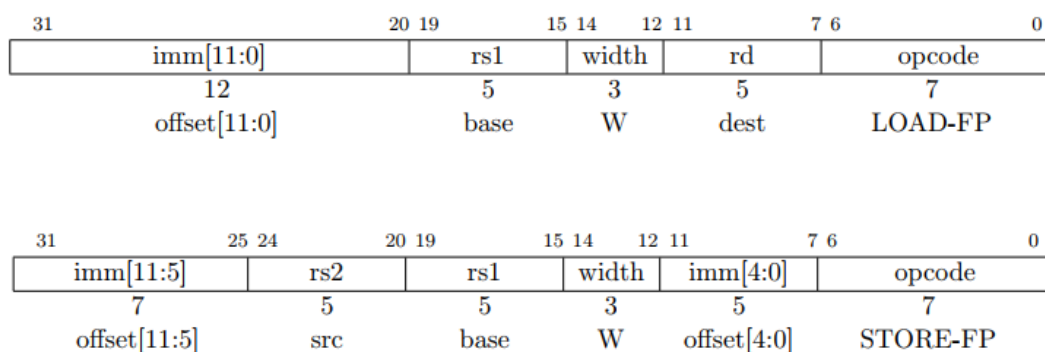


Figure 20. Single precision floating-point load/store instruction

4.1.2. Single-precision floating-point computational instructions

Computational instructions include the R-type commands FADD.S, FSUB.S, FMUL.S, FDIV.S and lastly FMIN.S/FMAX.S shown in Table 1. Except for FMIN.S and FMAX.S instructions, as the name implies, other instructions do basic four operations on floating-point numbers. FMIN.S and FMAX.S, on the other hand, saves the smaller or larger of the two floating-point numbers, respectively, to the destination register.

4.1.3. Single-precision floating-point move instructions

The floating-point move instructions, FMV.X.W and FMV.W.X shown in Table 1, are used to transfer contents of a floating-point register to a general purpose(integer) register or contents of a general purpose register to a floating-point register without changing or converting the data, respectively.

4.1.4. Single-precision floating-point compare instructions

There are 3 different instructions for comparison in F extension that are FEQ.S, FLT.S and FLE.S shown in Table 1, stands for equal, less than, less than or equal,

respectively. If the conditions are met, the integer destination register is written 1, otherwise 0.

4.1.5. Single-precision floating-point classify instruction

FCLASS.S is an instruction for classifying the contents of a floating-point register by writing a 10-bit mask to the integer register. The instruction classifies floating-point numbers by setting the corresponding bit in the integer register according to the Table 2 below. Classify instruction sets none of the floating-point exception flags. Instruction type and bit distributions given below in Table 3.

R_d bit	Meaning
0	is $-\infty$.
1	is a negative normal number.
2	is a subnormal number.
3	is -0 .
4	is $+0$.
5	is a positive subnormal number.
6	is positive normal number.
7	is $+\infty$.
8	is a signaling NaN.
9	is a quiet NaN.

Table 3. Classification Mask of floating-point numbers

4.1.6. Single-precision floating-point conversion instruction

In computational systems, it is often necessary to convert numbers between different formats, such as from integer to floating-point or vice versa. In RISC-V, this conversion is crucial for performance optimization in many algorithms and applications.

- **Integer to Floating-Point Conversion:**

In this operation, an integer is converted to its equivalent single-precision floating-point format. The integer is first represented as a binary number, and then the sign, exponent, and mantissa are computed. The integer value is normalized and the exponent is adjusted accordingly.

- **Floating-Point to Integer Conversion:**

This conversion involves extracting the exponent and mantissa from the floating-point representation and converting them to an integer format. The mantissa is scaled by the appropriate power of 2, and the result is rounded or truncated to fit within the target integer type.

For example, in RISC-V assembly, conversion instructions like `fcvt.s.w` can be used to convert from a 32-bit integer to a single-precision floating-point number, while `fcvt.w.s` converts a single-precision floating-point to a 32-bit integer.

4.1.7. Single-precision floating-point sign injection instruction

Sign injection is an operation used to manipulate the sign of a number without altering its magnitude. In floating-point arithmetic, this can be particularly useful when performing certain arithmetic operations where the sign needs to be adjusted, such as in conditional operations or when dealing with negative and positive values.

The **sign injection instruction** is useful for injecting the sign bit into a floating-point number, making it easier to handle sign changes without performing full floating-point arithmetic. In the context of RISC-V, this type of instruction can be used to efficiently handle signs in floating-point computations.

This operation is particularly useful for implementing operations like absolute value and sign flipping in floating-point arithmetic, which can be important in algorithms such as those used in digital signal processing and machine learning models.

4.1.8. Single-precision floating-point Fused Multiply Addition instruction

A Single-precision floating-point Fused Multiply-Add (FMA) instruction combines a multiplication and an addition operation into a single instruction, performing the computation $(A \times B) + C$ with only one final rounding step. This differs from separate multiply and add instructions, which would involve two rounding steps (one after the multiplication and another after the addition). The key advantage of FMA is increased precision and reduced latency, as intermediate rounding errors are minimized, leading to more accurate results, especially in iterative calculations common in scientific computing, signal processing, and graphics. Its atomic nature also means it can often be executed faster than two separate operations.

4.2. Overall System Design

The RISC-V processor is designed to support the **RV32IF** instruction set architecture, following a **5-stage pipelined architecture**:

- **Fetch (IF)**: This stage is responsible for fetching instructions from memory and incrementing the program counter (PC). The next PC value is determined based on the address provided by the **Branch Prediction Table**, ensuring efficient speculative execution and reducing pipeline stalls.
- **Decode (ID)**: In this stage, the fetched instruction is decoded into smaller components. Control signals are generated, source register values (either integer or floating-point) are read from the internal register files, and immediate values are extracted for use in subsequent pipeline stages.
- **Execute (EX)**: This stage pushes the read register values, immediate values, or forwarded data (in the case of data hazards) into the **ALU** or **FPU** for arithmetic or logical computation. Additionally, the EX stage handles comparisons and calculations for determining the next PC address in the case of branch or jump instructions, leveraging the **branch control unit**.
- **Memory (MEM)**: This stage performs memory access operations, either reading data from or writing data to memory. The effective memory address is calculated based on the decoded information from the ID stage, and the data comes from the ALU or directly from the instruction. In this stage, the memory access can be configured to handle either integer (I) or floating-point (F) data.
- **Writeback (WB)**: In the final stage, the results from the ALU/FPU or the data read by the **Load/Store Unit (LSU)**, or even the updated PC value (in the case of jump instructions), are written back to the appropriate integer or floating-point register files, depending on the executed instruction.

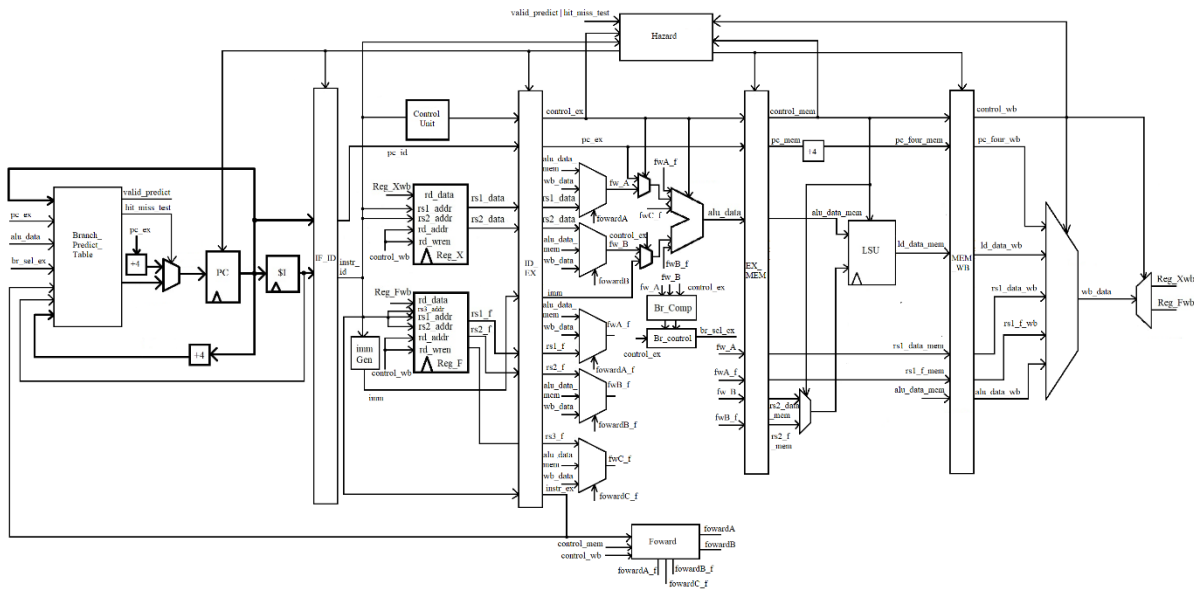


Figure 21. Pipeline Diagram of the RISC-V Processor with 'F' Extension

In this design, **hazards** are handled by the **Hazard Unit**. This unit compares the instructions in the EX, MEM, and WB stages to generate the necessary control signals, ensuring that the ALU in the EX stage receives the correct data forwarded from either the MEM or WB stages. Additionally, the Hazard Unit manages **stall signals** (to temporarily halt pipeline operation) and **flush signals** (to clear instructions from a stage) as needed when hazards are detected, ensuring correct instruction execution without data corruption.

The **Branch Prediction Table** assists in predicting the next PC address for branch and jump instructions, significantly improving overall system performance by reducing control hazards and minimizing pipeline stalls.

Moreover, a dedicated **Forwarding Unit** is implemented to handle data forwarding from later pipeline stages to the EX stage, particularly when the destination register has not yet been updated — covering both integer (I) and floating-point (F) instructions — thereby minimizing the impact of data hazards.

The processor supports the following categories of instructions:

- **Register-to-register operations:** ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND, FADD.S, FSUB.S, FMUL.S, FDIV.S, FMV.X.W, FMV.W.X, FCVT.W.S, FCVT.W.U.S, FCVT.S.W, FCVT.S.W.U, FSGNJ.S, FSGNJX.S, FSGNJN.S, FLT.S, FLE.S, FEQ.S, FMAX.S, FMIN.S, FMADD.S, FMSUB.S, FNMADD.S, FNMSUB.S.

- **Immediate operations:** ADDI, SLLI, SLTI, SLTIU, XORI, SRLI, SRAI, ORI, ANDI
- **Load instructions:** LW, FLW
- **Store instructions:** SW, FSW
- **Branch instructions:** BEQ, BNE, BLT, BGE, BLTU, BGEU
- **Jump instructions:** JAL, JALR
- **Upper immediate operations:** AUIPC, LUI

4.3. Detailed System Design

4.3.1. Program Counter

The **Program Counter (PC)** is fundamentally a **32-bit register** that functions as the program's address counter. On every rising edge of the clock signal, the value stored in this register is updated.

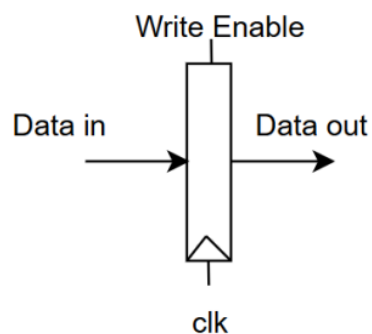


Figure 22. Program counter

Inputs	Output
Data In: 32-bit input data bus. Clk: Rising-edge triggered clock signal.	Data Out: 32-bit output data bus.

Table 4. Input/Output of Program Counter Block

Operation of the Program Counter (PC): On each rising edge of the clock signal, the PC updates its value such that Data Out = Data In. At all other times (i.e., when no rising edge occurs), Data Out remains unchanged, holding the previously stored value.

4.3.2. Register File

The register file consists of 32 registers, each 32 bits wide.

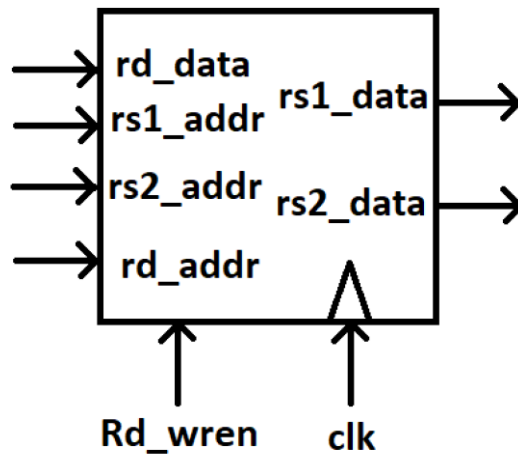


Figure 23. Register File Block

Inputs	Outputs
<p>Rd_data: 32-bit, contains the data to be written into a register.</p> <p>Rd_addr, rs1_addr, rs2_addr: 5-bit each, respectively the addresses of the register to write and the two registers to read.</p> <p>Clk: rising-edge clock signal.</p> <p>Rd_wren: 1-bit, the write-enable signal.</p>	<p>Data1, data2: 32-bit, the values read from the two addressed registers.</p>

Table 5. Inputs/Outputs of Register File

Operation: The registers are accessed via 5-bit addresses, with data1 = R[rs1], data2 = R[rs2], and R[rsW] = dataW only if RegWEn = 1. Writing occurs only on the rising edge of the clock, while reading behaves like a combinational system. Additionally, there is a separate register block dedicated to F, which is structurally the same as the I register

block but differs in the data written, depending on whether the instruction is an I-type or F-type.

4.3.3. Immediate Generator

The immediate generator produces the immediate number from an instruction.

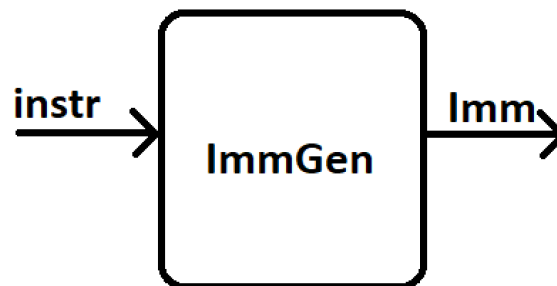


Figure 24. Immediate Generator Block

Inputs	Outputs
Inst[31:0]: the instruction.	Imm: 32-bit, contains the immediate value.

Table 6. Inputs/Outputs of Immediate Generator

Operation: The ImmGen works as a combinational block. For specific instruction types — I-type (e.g., addi, lw, jalr), S-type (e.g., sw), B-type (branch instructions), U-type (e.g., lui, auipc), J-type (jal), or floating-point loads/stores (flw, fsw) — it generates the appropriate immediate number from the input instruction.

4.3.4. Branch Comparator

The branch comparator serves branch instructions by comparing two values.

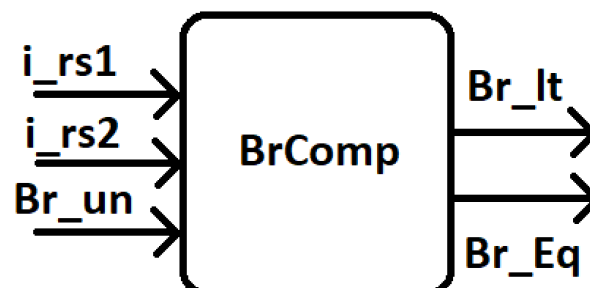


Figure 25. Branch Comparator Block

Inputs	Outputs
Rs1_i, rs2_i: 32-bit, containing the two register values fetched from the register file. BrUn: 1-bit, indicates whether the branch compares unsigned numbers.	BrEq: 1-bit, signals whether the two registers are equal. BrLt: 1-bit, signals whether the first register is less than the second.

Table 7. Inputs/Outputs of Branch Comparator

Operation:

If BrUn = 1, the comparator compares the two registers as unsigned numbers.

- If $rs1 == rs2$: BrEq = 1, BrLt = 0.
- If $rs1 < rs2$: BrEq = 0, BrLt = 1.
- Otherwise: both BrEq and BrLt are 0.

4.3.5. Branch Control

This block determines whether a branch or jump condition is satisfied.

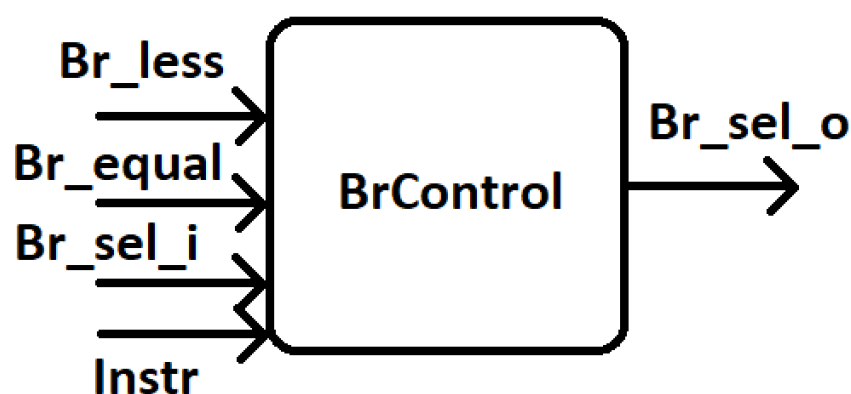


Figure 26. Branch Control Block

Inputs	Outputs
Br_sel_i: 1-bit, indicates the presence of a branch or jump instruction, sourced from pc_sel in the control unit. Br_less: 1-bit, signals that the first register is less than the second. Br_equal: 1-bit, signals that the two	Br_sel_o: 1-bit, signals whether the branch condition is satisfied.

registers are equal.

Instr: 32-bit, contains the instruction data.

Table 8. Inputs/Outputs of Branch Control

Operation: When a branch instruction is present and the comparison fails, Br_sel_o = 0. When the comparison succeeds, Br_sel_o = 1. For jump instructions, Br_sel_o is always 1.

4.3.6. Arithmetic Logic Unit (ALU)

The ALU performs operations like +, -, <<, >>, >>>, ^, &, |.

alu_op	Description (R-type)	Description (I-type)
ADD	$rd = rs1 + rs2$	$rd = rs1 + imm$
SUB	$rd = rs1 - rs2$	n/a
SLT	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
SLTU	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
XOR	$rd = rs1 \oplus rs2$	$rd = rs1 \oplus imm$
OR	$rd = rs1 \vee rs2$	$rd = rs1 \vee imm$
AND	$rd = rs1 \wedge rs2$	$rd = rs1 \wedge imm$
SLL	$rd = rs1 \ll rs2[4 : 0]$	$rd = rs1 \ll imm[4 : 0]$
SRL	$rd = rs1 \gg rs2[4 : 0]$	$rd = rs1 \gg imm[4 : 0]$
SRA	$rd = rs1 \ggg rs2[4 : 0]$	$rd = rs1 \ggg imm[4 : 0]$

Figure 27. Operations ALU performs

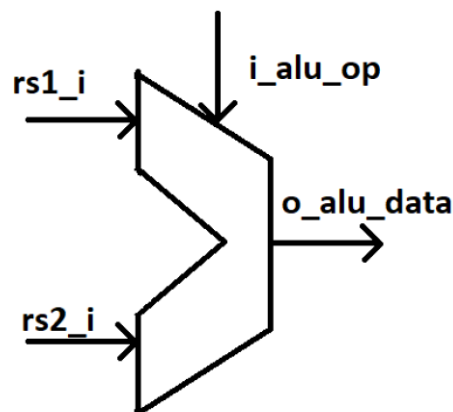


Figure 28. ALU Block

Inputs	Outputs
Rs1_i, rs2_i: 32-bit, containing data from	o_alu_data: 32-bit, contains the result of

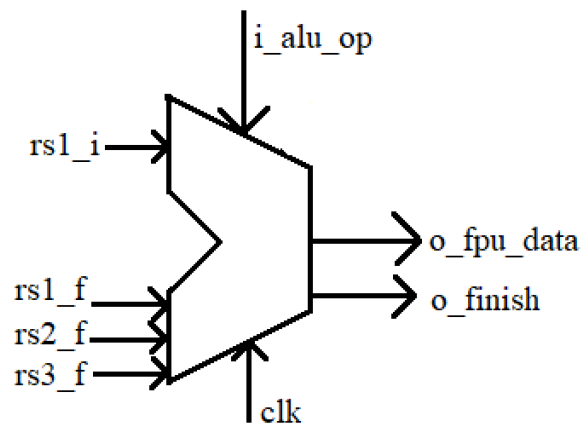
the I-type register file, or the PC value, or an immediate value from ImmGen.	the operation.
I_alu_op: 4-bit, selects the operation to perform.	

Table 9. Inputs/Outputs of ALU

Operation: For each i_alu_op signal, the ALU performs the corresponding operation (add, subtract, shift, etc.) as shown in the figure 20, and the result is output to o_alu_data.

4.3.7. Floating Point Unit (FPU)

The FPU handles floating-point single-precision operations as shown in the table 2.

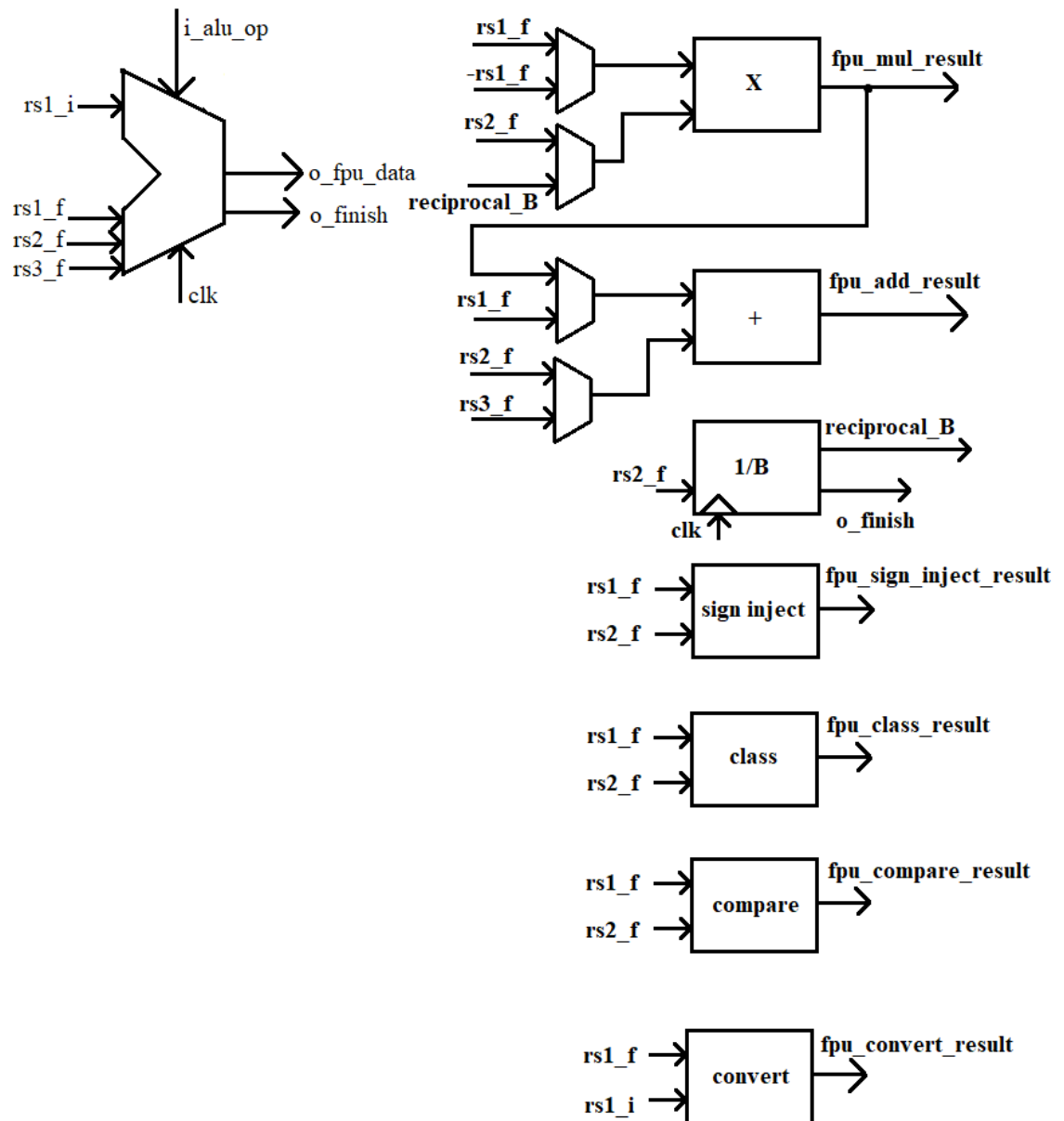
**Figure 29. FPU Block**

Inputs	Outputs
Clk: rising-edge clock signal. Rs1_f, rs2_f, rs3_f: 32-bit, data from the F-type register file. Rs1_i: 32-bit, data from the I-type register file. I_alu_op: 5-bit, selects the floating-point	o_fpu_data: 32-bit, contains the result of the floating-point operation. o_finish: 1-bit, signal the division has completed.

operation to perform.	
-----------------------	--

Table 10. Inputs/Outputs of FPU

Operation: For each `i_alu_op` signal, the FPU performs the corresponding floating-point operation (as shown in the design diagram) and outputs the result to `o_fpu_data`.

**Figure 30. Inside FPU Module.**

4.3.8. ALU-FPU

This block switches the control signal output between I-type and F-type results.

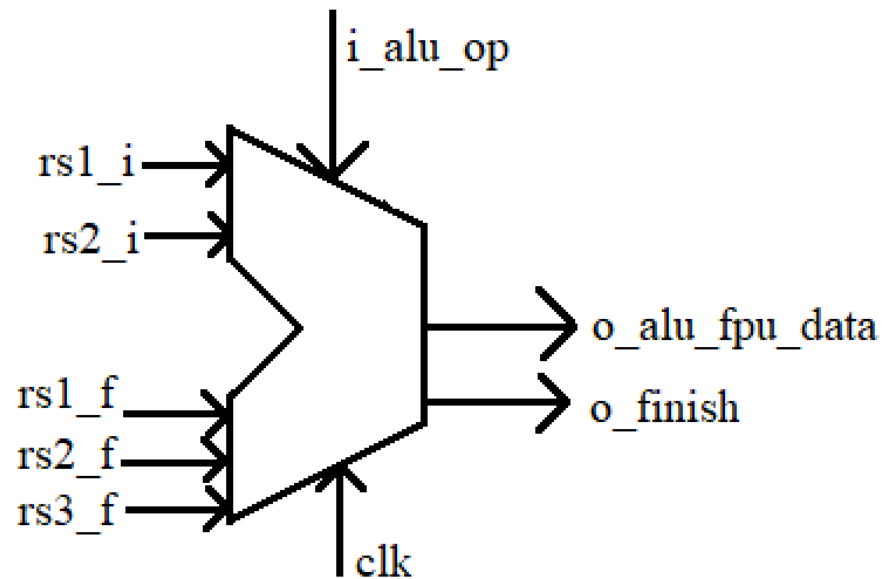


Figure 31. ALU-FPU Block

Inputs	Outputs
<p>Clk: rising-edge clock signal.</p> <p>Rs1_i, rs2_i: 32-bit, data from the I-type register file, the PC, or ImmGen.</p> <p>Rs1_f, rs2_f, rs3_f: 32-bit, data from the F-type register file.</p> <p>I_alu_op: 5-bit, selects the operation.</p>	<p>o_alu_fpu_data: 32-bit, contains the final result.</p> <p>o_finish: 1-bit, signal the division has completed.</p>

Table 11. Inputs/Outputs of ALU-FPU

Operation: For each i_alu_op signal, if the instruction is a float-type, the result comes from o_fpu_data; otherwise, it defaults to o_alu_data.

4.3.9. Control Unit

The control unit generates the control signals to orchestrate the operation of all other blocks.

instr	pc_sel	br_un	rd_wren_I	op_a_sel	op_b_sel	alu_op	mem_wren	func	wb_sel	lsu_sel	rd_wren_F	reg_sel
(R-R) op_I	*	*	1	R	R	op_I	Read	*	alu_data	*	0	I
(R-R) op_F	*	*	0	*	*	op_F	Read	*	alu_data	*	1	F
fcvt.w.s/fcvt.wu.s	*	*	1	*	*	op_F	Read	*	alu_data	*	0	I
fmv.x.w	*	*	1	*	*	op_F	Read	*	rs1_f	*	0	I
fcvt.s.w/fcvt.s.wu	*	*	0	R	*	op_F	Read	*	alu_data	*	1	F
fmv.w.x	*	*	1	*	*	op_F	Read	*	rs1_data	*	0	I
flw	*	*	0	*	*	*	Read	word	lsu_data	*	1	F
fsw	*	*	0	*	*	*	Write	word	*	rs2_f	0	I
Imm op_I	*	*	1	R	Imm	op_I	Read	*	alu_data	*	0	I
lw	*	*	1	R	Imm	*	Read	word	lsu_data	*	0	I
sw	*	*	0	R	Imm	*	Write	word	*	rs2_data	0	I
beq/bne/blt/bge	1	0	0	PC	Imm	*	Read	*	*	*	0	I
bltu/bgeu	1	1	0	PC	Imm	*	Read	*	*	*	0	I
jal	1	*	1	PC	Imm	*	Read	*	PC+4	*	0	I
jalr	1	*	1	R	Imm	*	Read	*	PC+4	*	0	I
lui/auipc	*	*	1	PC	Imm	op_I	Read	*	alu_data	*	0	I

Figure 32. Control Unit True Table

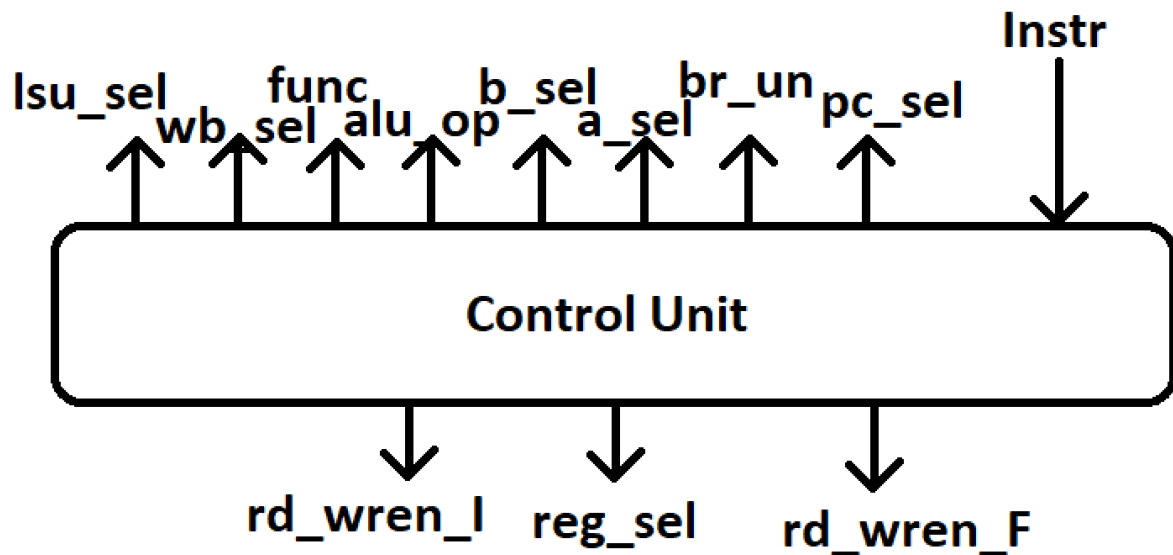


Figure 33. Control Unit Block

Inputs	Outputs
instr: 32-bit, the decoded instruction, which helps generate the necessary control signals.	Pc_sel: 1-bit, set to 1 for branch or jump instructions. Br_un: 1-bit, set to 1 for unsigned branch instructions.

	<p>Rd_wren_I: 1-bit, enables read/write on the I-type register file.</p> <p>Op_a_sel: 1-bit, selects between Reg and PC.</p> <p>Op_b_sel: 1-bit, selects between Reg and Imm.</p> <p>Alu_op: 5-bit, selects the ALU or FPU operation.</p> <p>Mem_wren: 1-bit, enables read/write to LSU.</p> <p>Func: 3-bit, selects between word, half-word, or byte access in LSU.</p> <p>Wb_sel: 3-bit, selects the data to write back to the register.</p> <p>Lsu_sel: 1-bit, selects between rs2_data (for sw) or rs2_f (for fsw).</p> <p>Rd_wren_F: 1-bit, enables read/write on the F-type register file.</p> <p>Reg_sel: 1-bit, selects which register file (I or F) the data is written back to.</p>
--	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 12. Inputs/Outputs of Control Unit

4.3.10. Hazard unit

Hazard module is designed to detect pipeline hazards in a 5-stage pipelined RISC-V processor and generate control signals to handle them. Specifically, it identifies load-use hazards and branch hazards, and outputs control signals to stall, flush, or clear specific pipeline stages.

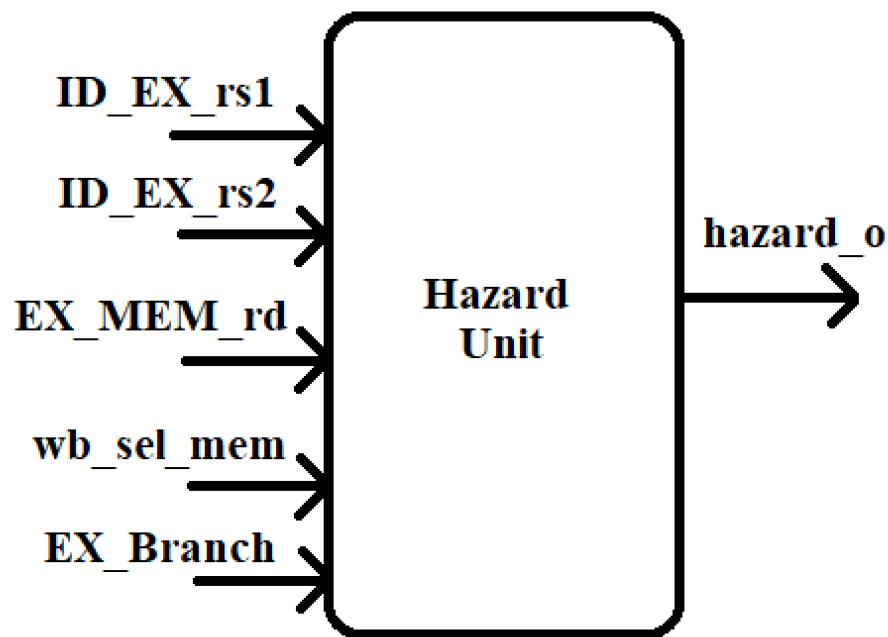


Figure 34. Hazard Unit Block.

Inputs	Outputs
<p>ID_EX_rs1 5 bits Source register 1 from the ID/EX stage (current instruction's rs1).</p> <p>ID_EX_rs2 5 bits Source register 2 from the ID/EX stage (current instruction's rs2).</p> <p>EX_MEM_rd 5 bits Destination register (rd) from the EX/MEM stage (instruction in MEM stage).</p> <p>wb_sel_mem 3 bits Write-back select signal in the MEM stage — identifies if the instruction in MEM is a load (2'b10).</p> <p>EX_branch 1 bit Signal indicating whether the current instruction in EX is a</p>	<p>hazard_o 8 bits Control output: {PC / IF_ID / ID_EX / EX_MEM / MEM_WB} → bit groups controlling stall (01), flush (11), or clear (00). Each group specifies what to do for that stage.</p>

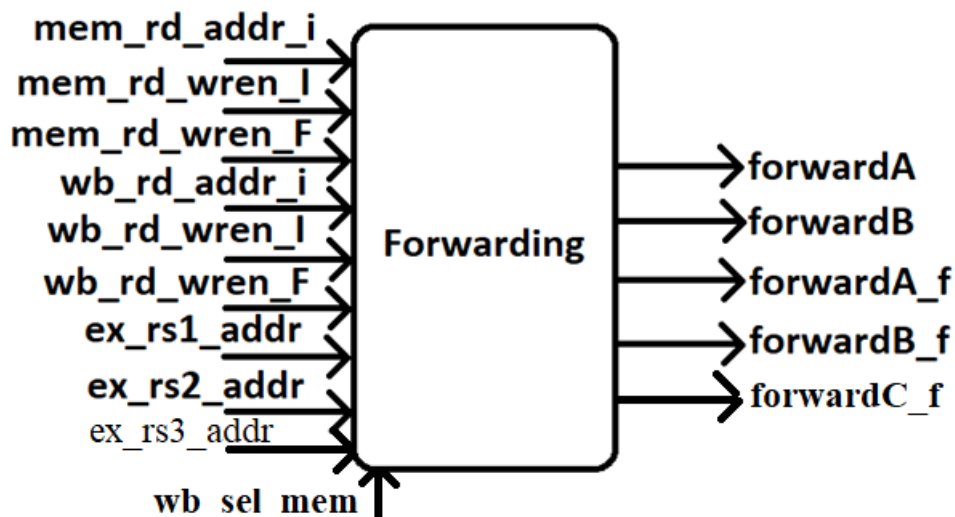
branch/jump.	
--------------	--

Table 13. Inputs/Outputs of Hazard Unit.

- Branch in EX Flush IF_ID and ID_EX
- Load-use in MEM stage Stall PC, IF_ID, ID_EX; flush EX_MEM
- No hazard No action (normal operation)

4.3.11. Forwarding

The forward module handles data forwarding (also called bypassing) in a pipelined processor. It detects if a source register (rs1 or rs2 in the EX stage) needs to get its data forwarded from a later pipeline stage (MEM or WB), avoiding pipeline stalls due to data hazards. This module separately handles integer (I) and floating-point (F) forwarding.

**Figure 35. Forwarding Block.**

Inputs	Outputs
mem_rd_addr_i 5 bits Destination register (rd) from the MEM stage. If 0, no forwarding.	forwardA_o 2 bits Forwarding control for rs1 integer.
mem_rd_wren_I_i 1 bit Write-enable for integer register in MEM stage.	forwardB_o 2 bits Forwarding control for rs2 integer.

<p>1 → enable forwarding.</p> <p>mem_rd_wren_F_i 1 bit Write-enable for floating-point register in MEM stage. 1 → enable forwarding.</p> <p>wb_rd_addr_i 5 bits Destination register (rd) from the WB stage. If 0, no forwarding.</p> <p>wb_rd_wren_I_i 1 bit Write-enable for integer register in WB stage. 1 → enable forwarding.</p> <p>wb_rd_wren_F_i 1 bit Write-enable for floating-point register in WB stage. 1 → enable forwarding.</p> <p>ex_rs1_addr_i 5 bits Source register 1 (rs1) address from the EX stage (current executing instruction).</p> <p>ex_rs2_addr_i 5 bits Source register 2 (rs2) address from the EX stage (current executing instruction).</p> <p>ex_rs3_addr_i 5 bits Source register 3 (rs3) address from the EX stage (current executing instruction).</p> <p>Wb_sel_mem 3 bits To Know if the current instruction is Load (3'b101).</p>	<p>forwardA_f_o 2 bits Forwarding control for rs1 floating-point.</p> <p>forwardB_f_o 2 bits Forwarding control for rs2 floating-point.</p> <p>forwardC_f_o 2 bits Forwarding control for rs3 floating-point.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 14. Inputs/Outputs of Forwarding.

Operation: The module checks for match conditions:

- If the MEM stage has a write-enabled register (mem_rd_wren_*) whose rd matches rs1 or rs2 → Forward from MEM stage (2'b01).

- Else if the WB stage has a write-enabled register (wb_rd_wren_*) whose rd matches rs1 or rs2 \rightarrow Forward from WB stage (2' b10).
- If neither matches \rightarrow No forwarding (2' b00).

4.3.12. Branch Predict Table

The predict_table module implements a branch prediction table with:

- A 2-bit saturating counter predictor (for dynamic prediction)
- A tag array (stores predicted target addresses)

It improves pipeline performance by speculatively predicting the next PC (nxt_pc_F_o) when encountering a branch or jump instruction, reducing stalls and misprediction penalties.

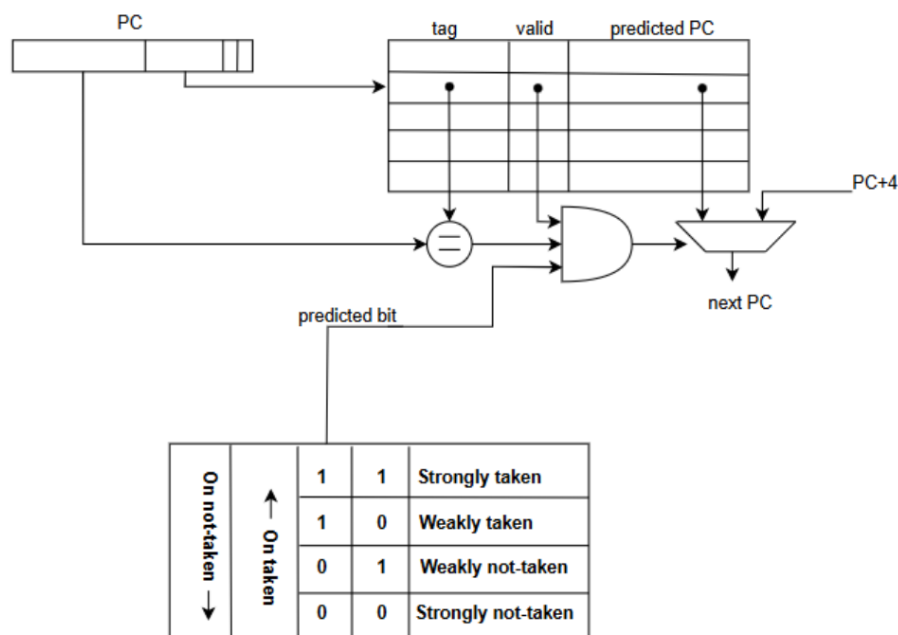


Figure 36. Structure of Branch Predict Table Block.

Whether the predicted PC value is used or not is shown in the diagram below.

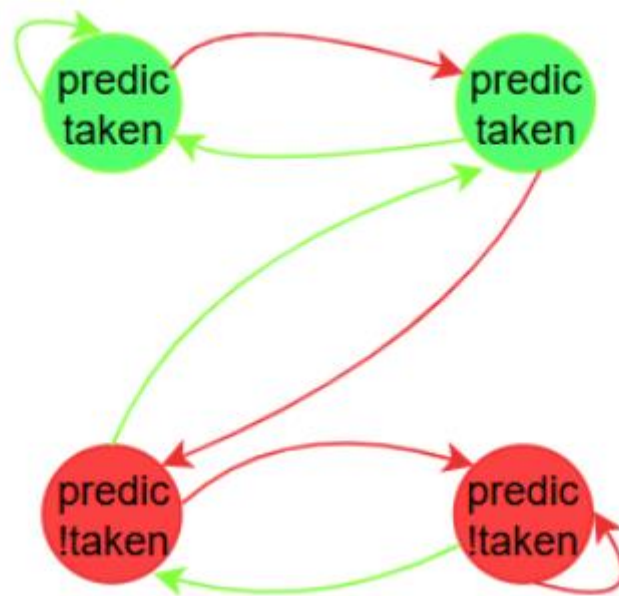


Figure 37. Predict bit diagram.

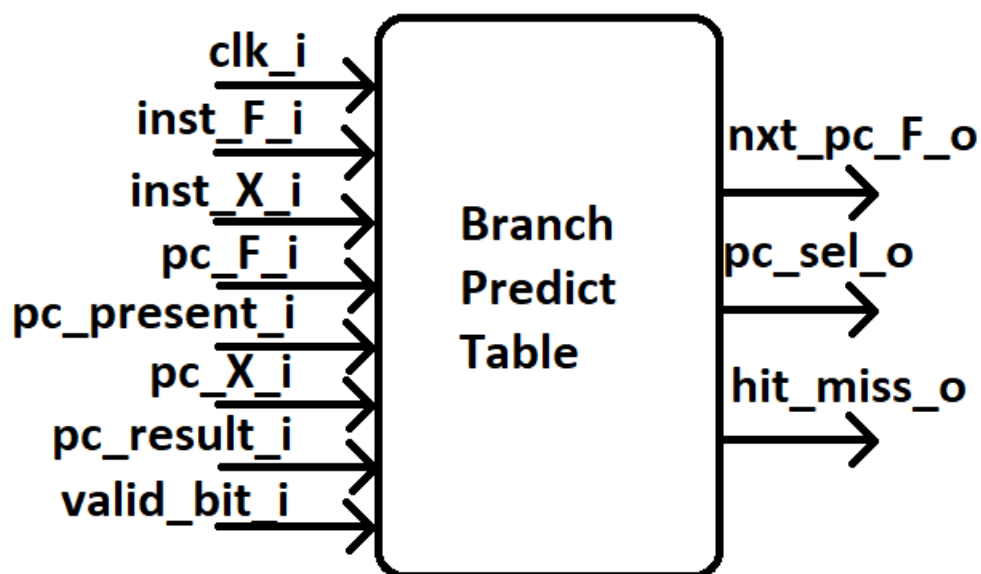


Figure 38. Branch Predict Table Block.

Inputs	Outputs
clk_i 1 bit Clock signal.	nxt_pc_F_o 32 bits Predicted next PC value for the Fetch stage.
inst_F_i 32 bits Instruction in Fetch (IF) stage — to check if it's a branch or jump.	pc_sel_o 1 bit Select signal: 1 → take

<p>inst_X_i 32 bits Instruction in Execute (EX) stage — to update predictor/tag if it's a branch or jump.</p> <p>pc_F_i 32 bits Program counter in Fetch stage.</p> <p>pc_present_i 13 bits Index (bits [12:0]) for current PC in the predictor/tag arrays.</p> <p>pc_X_i 13 bits Index (bits [12:0]) for PC in Execute stage (used to update predictor/tag arrays).</p> <p>pc_result_i 32 bits Actual branch/jump target calculated at Execute stage.</p> <p>valid_bit_i 1 bit Branch taken signal — tells if branch was actually taken (from EX stage).</p>	<p>resolved PC (pc_result_i), 0 → stay with predicted PC.</p> <p>hit_miss_o 1 bit Indicates if current prediction was a misprediction (1 = miss, 0 = hit). Calculated when predictor says taken but actual result was not.</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 15. Inputs/Outputs of Branch Predict Table.

5. Calculator Application On FPGA Using RISC-V IF

5.1. Overview

The calculator application implemented on FPGA utilizing the RISC-V 32IF architecture allows for efficient arithmetic operations with the following structured input and output phases. The system employs an intuitive interface, where users can enter numerical values through a keypad and view the results on an LCD screen. The calculator supports a maximum input of 16 characters per number, including a decimal point ".".

Input Phases and Transitions

The application follows a clearly defined phase structure, with each phase facilitating specific user actions. The transitions between these phases are determined by the user's inputs and adhere to the following sequence:

1. First Number Entry

The user begins by entering the first number using the keypad. This phase remains active until the user inputs the first operator (either +, -, *, or /). The system allows the input of up to 16 characters for this first number, including a single decimal point. Once the operator is entered, the system transitions to the **Operator Phase**.

2. Operator Phase

Following the entry of the first number and an operator, the system enters the **Operator Phase**. During this phase, the user may choose any of the four available arithmetic operators: addition (+), subtraction (-), multiplication (*), or division (/). The system prohibits further entry of operators once a valid operator has been selected, as well as the entry of the decimal point "." or the equals sign "=". The transition from this phase to the next occurs upon entering the second number.

3. Second Number Entry

In this phase, the user inputs the second number, following similar constraints to the first number entry. The second number may include up to 16 characters, with the decimal point "." allowed, but only once per number. Operators are restricted during this phase, and any further operator input is ignored. The phase concludes when the

user presses the equals sign "=" to finalize the calculation. This triggers the transition to the **Result Phase**.

4. Result Phase

In the **Result Phase**, the result of the arithmetic operation is displayed on the LCD screen. The result is shown with a maximum length of 16 characters, ensuring that large numbers or results with excessive decimal places are appropriately truncated or rounded for display. The system then resets, preparing for the next calculation.

System Constraints

- **Number Entry:** Both the first and second numbers can contain up to 16 characters, which includes digits (0-9), a decimal point (".") for floating-point numbers, and ensures the appropriate handling of decimal inputs. The decimal point can only appear once per number to maintain mathematical correctness.
- **Operator Input:** The system allows the selection of four basic arithmetic operations (+, -, *, /) but prevents any further entry of operators once the first operator is chosen.
- **Result Display:** After completing the second number entry and applying the selected arithmetic operation, the result is displayed, ensuring that the output fits within the 16-character limit of the display.

How we calculate the number using RISC-V

In this approach, each number entered via the keypad, including the first number, operator, and second number, is stored in a stack. The stack is then traversed until the first decimal point (".") is encountered, which marks the beginning of the fractional part. The first number is then added to the subsequent numbers, each multiplied by 10, until the entire fractional part is processed. This results in an integer register holding the total value of the fraction. The fraction is then divided by 10^n , where n is the number of digits in the fractional part, and stored in a floating-point register.

Next, the decimal part is calculated through a loop where the first number is multiplied by 10, and the next number is added repeatedly until the operator is reached. This process generates the total value of the decimal part, which is stored in another

floating-point register. The second number is computed by summing the fractional and decimal parts stored in their respective floating-point registers.

The operator is stored separately for later use. The first number undergoes a similar calculation to the second number. Based on the previously stored operator, the final result is computed and stored in another floating-point register. This floating-point register is then separated into two parts: the decimal and the fraction. The decimal part is obtained by converting the floating-point value to an integer, and the fractional part is calculated by subtracting the decimal part from the total.

Finally, the result is displayed on the LCD. The hexadecimal value of the result is first converted to decimal and shown on the screen. The fractional part is then multiplied by 1000, converted to an integer, and the result is again converted from hexadecimal to decimal for display on the LCD.

5.2. About Keypad

Once a number is entered, the system waits for an arithmetic operator input, such as addition (+), subtraction (-), multiplication (*), or division (/). The first number is temporarily stored in a floating-point register, and the system enters an intermediate state while waiting for the second number. The user then inputs the second number, which is also stored in a register. At this stage, the control logic determines the appropriate operation to be performed based on the selected operator.

The keypad interface is a fundamental input device for various embedded systems, including calculators, data entry systems, and control panels. It allows users to interact with the system by pressing keys that correspond to numerical or functional inputs. In this article, we explore two methods of implementing a keypad: (1) using software with a RISC-V processor to handle the calculations and (2) utilizing hardware (FPGA) to process the calculations, while the FPGA manages the keypad input.

Keypad Basics

A keypad is a matrix of switches organized in rows and columns. When a key is pressed, the row and column associated with that key are connected, allowing the system to detect the key press. Typically, keypads are used in embedded systems for user input, and their design requires both hardware and software integration.

Software-based Keypad Implementation with RISC-V

In a software-based keypad implementation, a RISC-V processor is responsible for detecting key presses and performing the required calculations. The process typically involves polling the keypad at regular intervals and checking the row and column states to identify the key that has been pressed.

Keypad Scanning Process

1. **Hardware Setup:** The keypad is connected to the GPIO (General Purpose Input/Output) pins of the RISC-V processor. The rows of the keypad are configured as output pins, and the columns are configured as input pins. A low voltage is sent to one row at a time, and the columns are read to detect which key is pressed.
2. **Software Logic:**
 - The software loops through each row, driving it low, and reads the columns to check if any key is pressed.
 - When a key is pressed, the row and column combination is mapped to a specific key value (e.g., number, operator).
 - Once the key is identified, the system performs the corresponding action (e.g., adding two numbers, displaying a number on the LCD).
3. **Debouncing:** Keypads can suffer from "bouncing," where a single key press is detected multiple times due to mechanical properties of the switches. Debouncing is implemented in software to ensure that only one key press is registered. This can be done by introducing a small delay after each key press detection and confirming that the key press is stable.
4. **Calculation:** After identifying the key, the RISC-V processor can perform the calculation based on the keypad inputs (e.g., number entry, operator selection). The processor executes the logic in software to compute the result, which is then displayed on the screen.

Software Flow

- **Keypad Initialization:** Configure the GPIO pins for keypad input and output.

- **Key Detection:** Poll the keypad periodically, scanning each row while checking the state of the columns.
- **Debounce Handling:** Implement a debounce mechanism to filter out spurious inputs.
- **Input Processing:** Once a valid key is pressed, the software performs the corresponding action (number entry, arithmetic operation).
- **Result Display:** After the operation is complete, the result is displayed on the output device (LCD or other display).

Advantages and Limitations

- **Advantages:**
 - Software provides flexibility in handling complex calculations and operations.
 - The RISC-V processor can be used to manage multiple tasks concurrently, making it suitable for applications beyond simple calculation.
- **Limitations:**
 - Software-based key scanning may be slower due to the need for regular polling of GPIO pins.
 - Debouncing must be handled in software, which adds overhead to the processor.

Hardware-based Keypad Implementation with FPGA

In contrast to the software-based approach, an FPGA-based implementation offloads the keypad scanning and calculation tasks to the FPGA hardware. The FPGA is a reconfigurable hardware platform, and it can directly handle the keypad input, perform calculations, and output the results with minimal latency.

FPGA Keypad Scanning Process

1. Hardware Setup:

- The keypad is connected to the FPGA's input/output pins. The FPGA is configured to read the rows and columns of the keypad.

- The FPGA can use hardware description languages (HDL), such as VHDL or Verilog, to implement the logic for scanning the keypad matrix.

2. Keypad Scanning Logic:

- The FPGA continuously drives each row one at a time and reads the state of the columns to identify which key is pressed.
- Once a key is detected, the FPGA can instantly process the input and trigger the corresponding operation.

3. Debouncing in Hardware:

- The FPGA can implement hardware-based debouncing by using flip-flops or counters. This ensures that key presses are stable before processing them.
- Debouncing can be handled as part of the FPGA logic without requiring software intervention.

4. Calculation and Result Output:

- The FPGA can perform calculations directly in hardware, using arithmetic logic units (ALUs) or custom-designed modules for addition, subtraction, multiplication, and division.
- The result can be displayed on the LCD or another output device connected to the FPGA.

FPGA Flow

- **Keypad Initialization:** Configure FPGA pins for keypad scanning.
- **Key Detection:** The FPGA continuously scans the keypad matrix, detecting key presses and performing calculations.
- **Debounce Handling:** Implement hardware debouncing logic using flip-flops or counters.
- **Calculation:** The FPGA directly performs the arithmetic operation based on the key pressed.

- **Result Display:** Output the result to the LCD or other display device.

Advantages and Limitations

- **Advantages:**
 - FPGA-based implementations provide real-time performance with low latency, as all tasks are handled in hardware.
 - Offloading key scanning and calculations to the FPGA frees up the processor for other tasks or reduces the overall system power consumption.
 - Hardware debouncing in FPGA ensures more reliable key press detection.
- **Limitations:**
 - FPGA programming requires a deep understanding of hardware description languages (VHDL or Verilog), which may be more complex than software-based approaches.
 - The FPGA implementation is typically less flexible than a software solution, as changes in functionality may require reprogramming the FPGA.

Comparison of Software and Hardware Approaches

Feature	Software (RISC-V) Implementation	Hardware (FPGA) Implementation
Keypad Scanning	Polling of GPIO pins in software	Continuous scanning in FPGA hardware
Debouncing	Implemented in software (with delays)	Implemented in hardware using flip-flops/counters
Calculation	Performed by RISC-V processor software	Performed by FPGA hardware directly
Latency	Higher latency due to software overhead	Lower latency due to hardware processing

Flexibility	Highly flexible, can easily handle complex operations	Less flexible, harder to modify after implementation
Complexity	Simpler to implement, but requires software for debouncing and calculation	More complex to implement, but faster and more efficient

Table 16. Comparison of Software and Hardware Approaches

Conclusion

The choice between implementing a keypad interface using software on a RISC-V processor or hardware on an FPGA depends on the specific needs of the application. Software-based implementations are more flexible and easier to modify but may incur higher latency due to the overhead of processing and polling. On the other hand, FPGA implementations provide faster, real-time performance and can handle debouncing and calculation directly in hardware, making them ideal for applications requiring low latency and high throughput. Both methods have their respective advantages and limitations, and the decision should be based on the trade-offs in terms of speed, flexibility, and complexity required for the application.

5.3. About LCD

In embedded systems, Liquid Crystal Displays (LCDs) are widely used to present output data in a human-readable form. The **CFAH1602B-TMC-JP** is a 16x2 character LCD module, capable of displaying up to 32 characters in a 16-character wide, 2-line format. This module, based on the HD44780 LCD controller, is a popular choice in many embedded projects due to its simplicity, low cost, and ease of interfacing. In this article, we explore the process of interfacing and implementing the **CFAH1602B-TMC-JP** LCD module on the **FPGA DE2** platform, detailing both the hardware and software aspects of the design.

Overview of the CFAH1602B-TMC-JP LCD Module

The **CFAH1602B-TMC-JP** is a 16x2 character LCD that utilizes a parallel interface, making it suitable for interfacing with various microcontrollers, FPGAs, and other embedded systems. Key features of the module include:

- **Display Configuration:** 2 rows of 16 characters (total 32 characters).

- **Controller:** HD44780, a widely used controller for character-based LCDs.
- **Interface:** 8-bit or 4-bit parallel interface for communication.
- **Backlight:** Available to illuminate the display for better visibility.
- **Voltage:** Operates typically at 5V, compatible with FPGA DE2's I/O voltage levels.

The module supports a range of operations, including character display, cursor positioning, and basic animation, making it an ideal choice for applications requiring simple text output.

FPGA DE2 Platform Overview

The **FPGA DE2** is a development board designed by Terasic, built around the **Cyclone II FPGA** from Intel (formerly Altera). It features a wide variety of peripherals, including switches, LEDs, a 7-segment display, and an LCD connector. The FPGA DE2 platform provides a comprehensive development environment for implementing custom digital logic circuits and interfacing with various peripherals.

The **Cyclone II FPGA** on the DE2 board is equipped with **I/O pins** that can be configured to communicate with the **CFAH1602B-TMC-JP** LCD module using either an 8-bit or 4-bit parallel interface. The FPGA allows users to design custom logic for controlling the LCD, providing flexibility in both hardware design and timing control.

Interfacing the CFAH1602B-TMC-JP LCD with FPGA DE2

The **CFAH1602B-TMC-JP** LCD module operates over a parallel interface that requires several control and data pins. Below is an overview of how to interface the LCD with the FPGA DE2:

Pin Connections

The **CFAH1602B-TMC-JP** typically has 16 pins, with specific functions as follows:

1. **VSS:** Ground (connected to FPGA ground).
2. **VDD:** Power supply (typically 5V, connected to the FPGA's 5V output).
3. **VO:** Contrast control (typically connected to a potentiometer or ground for default contrast).

4. **RS**: Register Select (used to select between data and command registers).
5. **RW**: Read/Write (controls whether the operation is read or write).
6. **E**: Enable (used to latch the data into the LCD).
7. **D0 to D7**: Data pins (8-bit data bus).
8. **A**: LED anode (for backlight, typically connected to 5V through a current-limiting resistor).
9. **K**: LED cathode (for backlight, connected to ground).

To communicate with the LCD, we need to set up the following connections on the FPGA DE2:

- **RS (Register Select)**, **RW (Read/Write)**, and **E (Enable)** can be connected to general-purpose I/O pins on the FPGA.
- **Data pins (D0 to D7)** can be connected to a set of 8 FPGA I/O pins.
- **VSS** and **VDD** are connected to ground and 5V, respectively.

Control Signals

The LCD module requires control signals to operate. These are:

- **RS (Register Select)**: Determines whether data or commands are being written to the LCD. When **RS** is LOW, the operation is a command (such as clearing the display), and when HIGH, the operation is a data write (displaying a character).
- **RW (Read/Write)**: Specifies whether the operation is read or write. Typically, this is set to LOW for write operations.
- **E (Enable)**: This signal triggers the LCD to latch the data present on the data pins. A HIGH pulse on **E** will latch the data or command.

Data Signals

The **D0 to D7** pins are used to send data or commands to the LCD. These are 8-bit data lines, but the LCD also supports a 4-bit mode to reduce the number of I/O pins required. For simplicity, we assume an 8-bit interface for this article.

LCD Commands and Control Operations

The **CFAH1602B-TMC-JP** uses the **HD44780** controller, which defines a set of commands for configuring the display, controlling the cursor, clearing the screen, and setting display modes. These commands are sent to the LCD via the data bus (8-bit or 4-bit mode) along with the necessary control signals (**RS**, **RW**, **E**) to distinguish between data and command writes.

LCD Command Structure

Commands are sent to the LCD module by toggling the **RS** (Register Select) pin. When **RS** is **LOW**, the data sent to the LCD is interpreted as a command. When **RS** is **HIGH**, the data is interpreted as display data (i.e., characters to be shown on the screen).

Commands are 8 bits wide, and each bit serves a specific function. Below are the most common commands that are used with the **HD44780** LCD controller (which applies to the **CFAH1602B-TMC-JP** as well):

1. **Function Set (0x38)**: This command is used to initialize the LCD. It specifies the interface type (8-bit or 4-bit), the number of lines (1 or 2), and character font. A common command is 0x38 to set the display in 8-bit mode, 2 lines, and 5x8 dots font.
 - **Command**: 0x38
 - **Description**: 8-bit interface, 2-line display, 5x8 dots font.
2. **Display Control (0x0C)**: This command turns the display on or off, and controls the cursor and blinking.
 - **Command**: 0x0C
 - **Description**: Turn on display, turn off cursor, turn off blinking.
3. **Clear Display (0x01)**: This command clears the display and resets the cursor position to the beginning of the first line.
 - **Command**: 0x01
 - **Description**: Clears the display.

4. **Return Home (0x02):** This command returns the cursor to the home position (start of the first line).
 - **Command:** 0x02
 - **Description:** Return cursor to home position.
5. **Entry Mode Set (0x06):** This command sets the entry mode, which defines how the cursor moves after each character is written (to the right or left).
 - **Command:** 0x06
 - **Description:** Set entry mode to increment cursor position, no shift.
6. **Cursor Shift (0x14, 0x18):** These commands shift the cursor position either to the left or right.
 - **Command:** 0x14 (shift right), 0x18 (shift left)
 - **Description:** Move the cursor one position to the left or right.
7. **Set DDRAM Address (0x80):** This command moves the cursor to a specific position on the display, given by the address.
 - **Command:** 0x80 + address (e.g., 0x80 for the first position, 0xC0 for the second line).
 - **Description:** Move cursor to the specified DDRAM address.

Command and Data Timing

When sending commands to the LCD, there is a timing sequence that must be followed to ensure proper operation. Typically, each command must be followed by a small delay, allowing the LCD to process the command. This delay depends on the operation being executed (e.g., clearing the screen takes longer than setting the cursor).

- **Command Delay:** Typically, the delay after sending a command is around **40 microseconds**. This delay may vary depending on the LCD module's configuration and the operation.

- **Data Delay:** Similar to the command delay, after sending data (a character to display), a small delay is usually required to ensure the character is correctly written to the LCD.

ASCII Data Handling

The **CFAH1602B-TMC-JP** LCD can display ASCII characters, which are sent as data to the LCD. Each character corresponds to an ASCII code, which is a 7-bit value representing a character in the standard ASCII table.

The **ASCII character set** includes both printable characters (letters, digits, punctuation) and control characters. For the **CFAH1602B-TMC-JP** LCD, only the printable characters are displayed, while control characters are used to manage the display and cursor operations.

ASCII Character Encoding

The **CFAH1602B-TMC-JP** LCD can display characters in the **ASCII** format. The ASCII values for characters are typically in the range from **0x20 (32 in decimal)** for the space character to **0x7E (126 in decimal)** for the tilde (~) character. Here's a brief overview of how ASCII characters are represented:

ASCII Code	Character	ASCII Code	Character
0x20	Space	0x30	0
0x21	!	0x31	1
0x22	"	0x32	2
...
0x41	A	0x61	a
0x42	B	0x62	b

ASCII Code	Character	ASCII Code	Character
...
0x7A	z	0x7E	~

Table 17. Overview of how ASCII characters are represented

The ASCII values of these characters are sent to the **CFAH1602B-TMC-JP** as data. For instance, to display the character "A", the ASCII code 0x41 is sent to the LCD's **Data Register**. The character will then appear on the display at the current cursor position.

Writing Data to the LCD

The process of writing data to the LCD involves sending an ASCII character to the LCD's **Data Register**. This is done as follows:

1. **Set RS to HIGH:** This tells the LCD that the data being sent is to be displayed as a character (not a command).
2. **Set RW to LOW:** This specifies that the operation is a write (not a read).
3. **Set E to HIGH:** This latches the data to the LCD.
4. **Send ASCII Data:** The corresponding ASCII value of the character is sent via the data bus (8-bit or 4-bit mode).
5. **Set E to LOW:** This ends the write cycle.

For example, to display the character "A", the **ASCII value 0x41** is written to the LCD, and the character will appear on the screen.

5.4. Flowchart For The Application

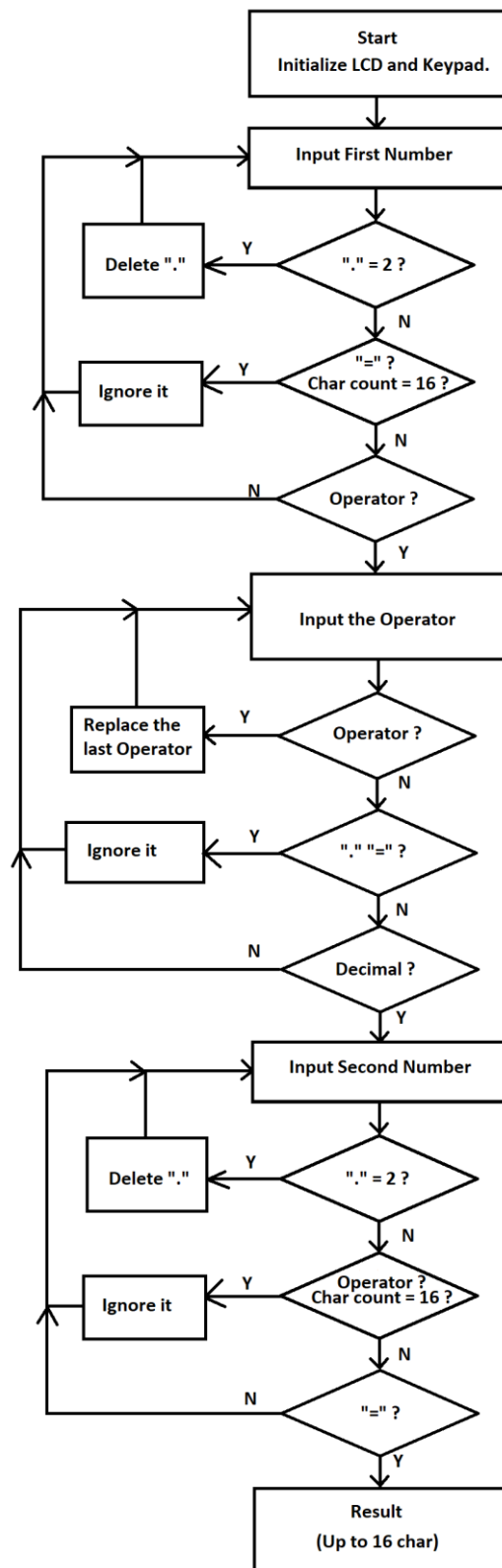


Figure 39. Overview Calculator Flowchart

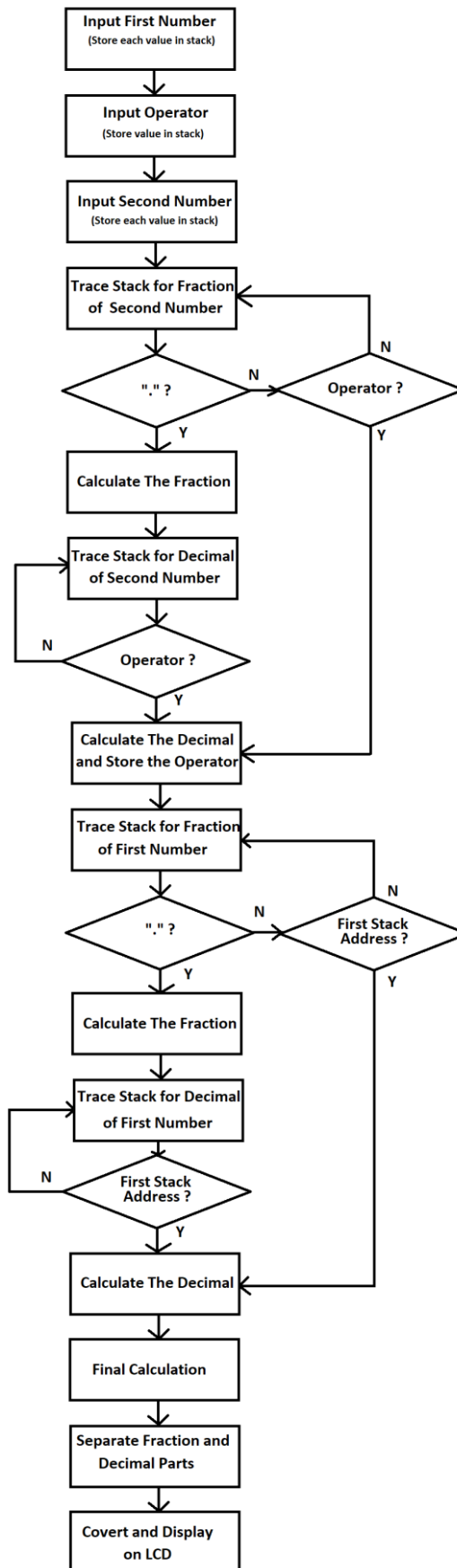


Figure 40. How we calculate the number using RISC-V

6. Verification and results

6.1. Verification for RISC-V IF

6.1.1. Waveform testbench on Modelsim/Questasim

6.1.1.1. Testing application on Questasim

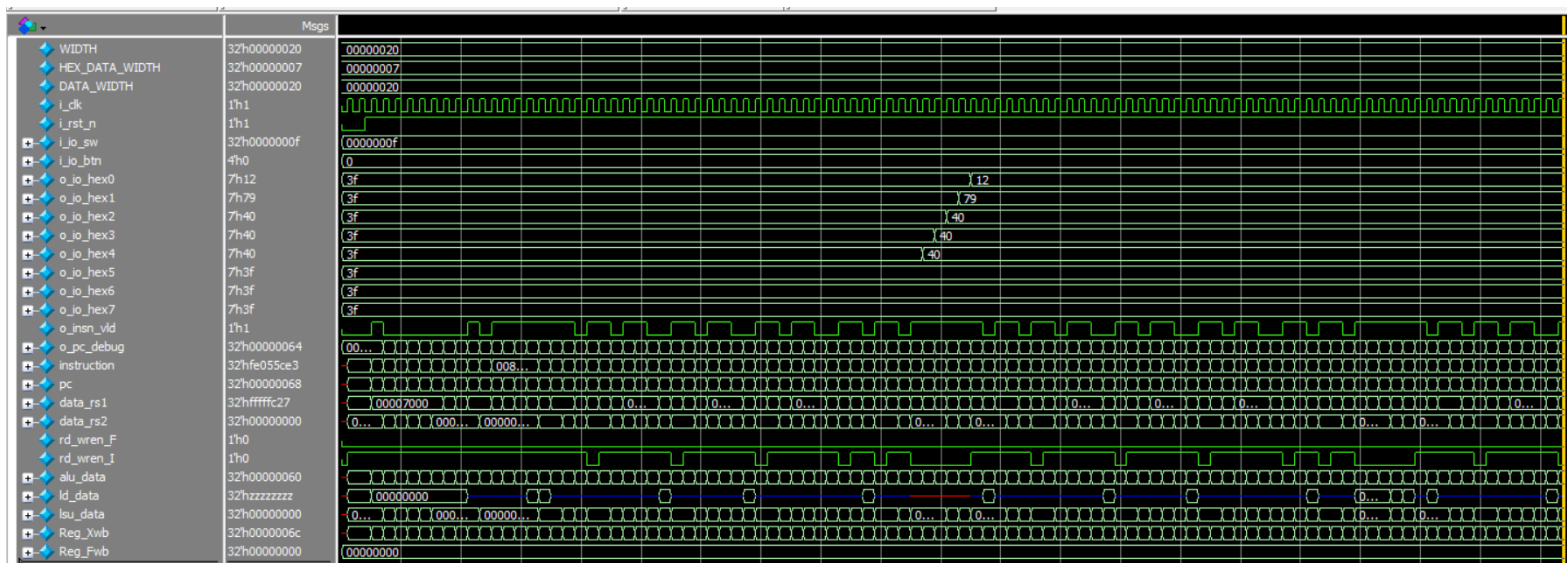


Figure 41. Waveform testbench on Questasim

Explain about the application:

This program is written in RISC-V assembly and aims to read a 16-bit value from a memory-mapped switch peripheral and display its decimal representation across five HEX displays (HEX0 to HEX4). The HEX displays are assumed to be memory-mapped I/O devices located at specific addresses. The program repeatedly reads the switch input and updates the HEX displays in real time.

Memory Address Mapping

The memory-mapped I/O addresses for the HEX displays and switches are organized as follows:

- 0x7020: HEX0 (Least significant digit)
- 0x7021: HEX1
- 0x7022: HEX2
- 0x7023: HEX3
- 0x7024: HEX4 (Most significant digit)

- 0x7800: Switch input (16-bit value)

This RISC-V assembly program demonstrates real-time conversion of binary input (from switches) into a human-readable decimal format displayed on HEX displays. It uses only basic arithmetic operations (addition and subtraction), avoiding complex division instructions, making it suitable for simple hardware environments such as FPGAs or educational RISC-V cores. This approach ensures compatibility and ease of implementation in embedded systems with limited instruction sets.

Explain the simulation result:

As we can see from the simulation, input switches have value 0x0000000F (15 in decimal). The output on HEX0 and HEX1 is 79 and 12, which are 5 and 1 on HEX LED display.

6.1.1.2. Testing Integer Instruction

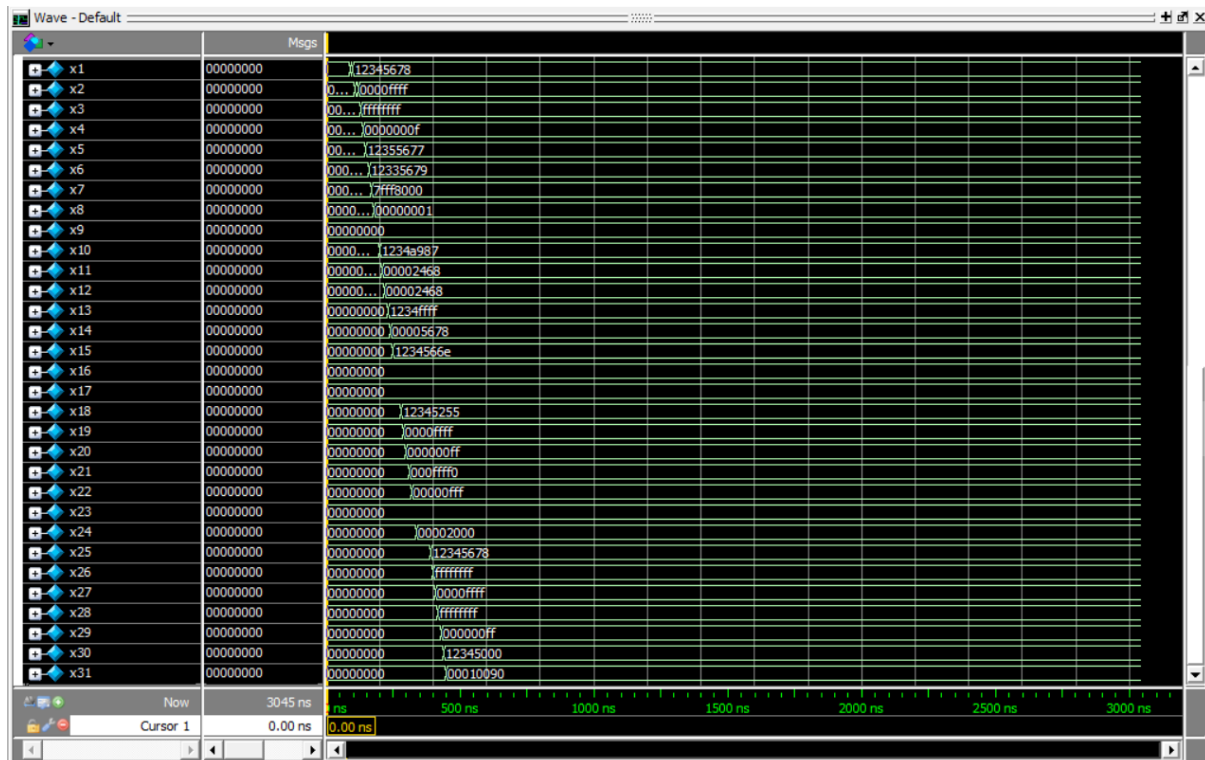


Figure 42. Register Result For The Instruction.

This RISC-V assembly program demonstrates the use of **basic arithmetic, logical, shift, immediate, load/store, and upper immediate** instructions. It manipulates values in general-purpose registers (x1–x31) to illustrate how each operation works at the instruction level.

1. Register Initialization:

li x1, 0x12345678 # Load x1 with 0x12345678

li x2, 0x0000FFFF # Load x2 with 0x0000FFFF

li x3, 0xFFFFFFFF # Load x3 with 0xFFFFFFFF (–1 in signed)

li x4, 0x0000000F # Load x4 with 15 (used as shift amount)

Registers are initialized with constants to be used in subsequent arithmetic and logic operations.

2. Arithmetic & Logical (Register-to-Register):

add x5, x1, x2 # Add: x5 = x1 + x2

```
sub x6, x1, x2    # Subtract: x6 = x1 - x2

sll x7, x2, x4    # Shift Left Logical: x7 = x2 << x4 (by 15 bits)

slt x8, x2, x1    # Set Less Than (signed): x8 = (x2 < x1) ? 1 : 0

sltu x9, x3, x2   # Set Less Than Unsigned: x9 = (x3 < x2) ? 1 : 0

xor x10, x1, x2   # Bitwise XOR: x10 = x1 ^ x2

srl x11, x1, x4   # Shift Right Logical: x11 = x1 >> 15

sra x12, x1, x4   # Shift Right Arithmetic: x12 = x1 >> 15 (sign-extended)

or  x13, x1, x2   # Bitwise OR: x13 = x1 | x2

and x14, x1, x2   # Bitwise AND: x14 = x1 & x2
```

These instructions perform arithmetic and logical operations between two registers and store the result in a destination register.

Supposed Result:

```
t0 (x5) = 0x12355677
t1 (x6) = 0x12335679
t2 (x7) = 0x7FFF8000
s0 (x8) = 0x00000001
s1 (x9) = 0x00000000
a0 (x10) = 0x1234A987
a1 (x11) = 0x00002468
a2 (x12) = 0x00002468
a3 (x13) = 0x1234FFFF
a4 (x14) = 0x00005678
```

3. Arithmetic & Logical (Register + Immediate)


```
addi x15, x1, -10    # Add Immediate:  $x15 = x1 - 10$ 

slti x16, x2, 100    # Set Less Than Immediate (signed)

sltiu x17, x3, 100   # Set Less Than Immediate Unsigned

xori x18, x1, 0x042D # XOR Immediate:  $x18 = x1 \wedge 0x042D$ 

ori  x19, x2, 0x04B4 # OR Immediate

andi x20, x2, 0x00FF # AND Immediate

slli x21, x2, 4      # Shift Left Logical Immediate

srli x22, x2, 4      # Shift Right Logical Immediate

srai x23, x3, 4      # Shift Right Arithmetic Immediate
```

These operations are similar to the previous group, but they use a constant (immediate) value instead of a second register. The shift instructions use immediate values for shift amounts.

Supposed Result:

```
a5 (x15) = 0x1234566E
a6 (x16) = 0x00000000
a7 (x17) = 0x00000000
s2 (x18) = 0x12345255
s3 (x19) = 0x0000FFFF
s4 (x20) = 0x000000FF
s5 (x21) = 0x000FFFF0
s6 (x22) = 0x00000FFF
s7 (x23) = 0xFFFFFFFF
```

4. Load/Store Instructions

```
li  x24, 0x2000    # Load address of 0x2000` into x24
```

sw x1, 0(x24) # Store word from x1 to memory at x24

sh x2, 4(x24) # Store halfword from x2 at offset 4

sb x3, 6(x24) # Store byte from x3 at offset 6

lw x25, 0(x24) # Load word from memory into x25

lh x26, 4(x24) # Load signed halfword

lhu x27, 4(x24) # Load unsigned halfword

lb x28, 6(x24) # Load signed byte

lbu x29, 6(x24) # Load unsigned byte

This section performs memory operations:

- sw, sh, sb: store data from a register into memory.
- lw, lh, lhu, lb, lbu: load data from memory into a register (with signed/unsigned variants).

Supposed Result:

s8 (x24) = 0x00002000

s9 (x25) = 0x12345678

s10 (x26) = 0xFFFFFFFF

s11 (x27) = 0x0000FFFF

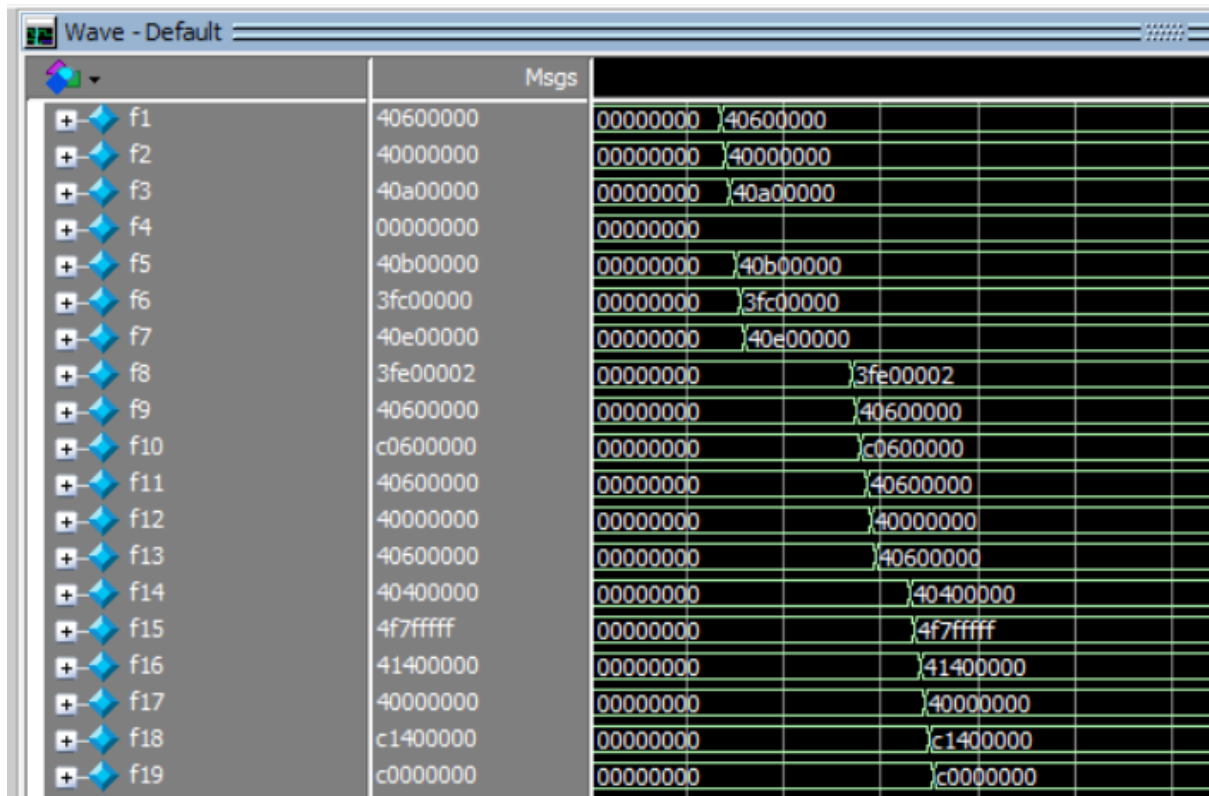
t3 (x28) = 0xFFFFFFFF

t4 (x29) = 0x000000FF

t5 (x30) = 0x12345000

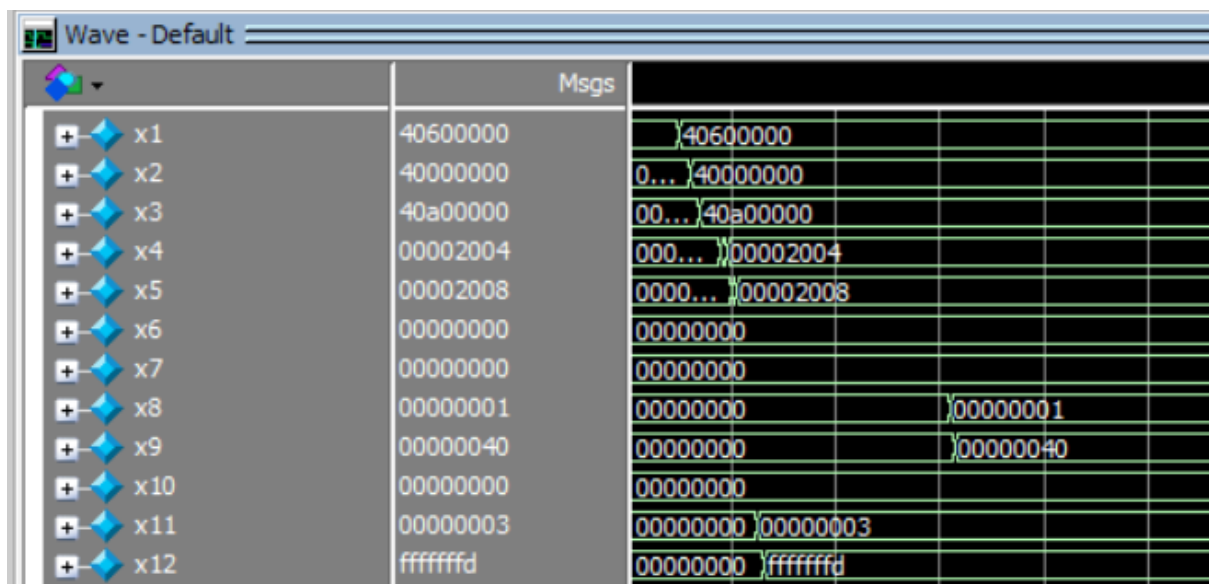
t6 (x31) = 0x00010090

6.1.1.3. Testing application on Questasim



		Msgs
f1	40600000	00000000 40600000
f2	40000000	00000000 40000000
f3	40a00000	00000000 40a00000
f4	00000000	00000000
f5	40b00000	00000000 40b00000
f6	3fc00000	00000000 3fc00000
f7	40e00000	00000000 40e00000
f8	3fe00002	00000000 3fe00002
f9	40600000	00000000 40600000
f10	c0600000	00000000 c0600000
f11	40600000	00000000 40600000
f12	40000000	00000000 40000000
f13	40600000	00000000 40600000
f14	40400000	00000000 40400000
f15	4f7fffff	00000000 4f7fffff
f16	41400000	00000000 41400000
f17	40000000	00000000 40000000
f18	c1400000	00000000 c1400000
f19	c0000000	00000000 c0000000

Figure 43. F-extension Testing on Modelsim.



		Msgs
x1	40600000	40600000
x2	40000000	0... 40000000
x3	40a00000	00... 40a00000
x4	00002004	000... 00002004
x5	00002008	0000... 00002008
x6	00000000	00000000
x7	00000000	00000000
x8	00000001	00000000 00000001
x9	00000040	00000000 00000040
x10	00000000	00000000
x11	00000003	00000000 00000003
x12	fffffffd	00000000 fffffffd

Figure 44. Value Data For F-extension Testing on Modelsim.

li x1, 0x40600000 #3.5 in float

li x2, 0x40000000 #2.0 in float

li x3, 0x40a00000 #5.0 in float

li x30, 0x2000

li x4, 0x2004

li x5, 0x2008

sw x1, 0(x30)

sw x2, 0(x4)

sw x3, 0(x5)

li x11, 0x00000003 #3 in integer

li x12, 0xFFFFFFFF # -3 in integer

flw f1, 0(x30)

flw f2, 0(x4)

flw f3, 0(x5) # Load a third float value

fadd.s f5, f1, f2 #expected result is 0x40B00000 (5.5 in float)

fsub.s f6, f1, f2 #expected result is 0x3fc00000 (1.5 in float)

fmul.s f7, f1, f2 #expected result is 0x40e00000 (7.0 in float)

fdiv.s f8, f1, f2 #expected result is 0x3fe00000 (1.75 in float)

fsgnj.s f9, f1, f2 #expected result is 0x40600000 (3.5 in float)

fsgnfn.s f10, f1, f2 #expected result is 0xc0600000 (-3.5 in float)

fsgnjx.s f11, f1, f2 #expected result is 0x40600000 (3.5 in float)

fmin.s f12, f1, f2 #expected result is 0x40000000 (2.0 in float)

fmax.s f13, f1, f2 #expected result is 0x40600000 (3.5 in float)

feq.s x6, f1, f2 #expected result is 0x00000000 (false)

flt.s x7, f1, f2 #expected result is 0x00000000 (false)

fle.s x8, f2, f1 #expected result is 0x00000001 (true)

fclass.s x9, f1 #expected result is 0001000000 (normal)

fcvt.w.s x11, f1 #expected result is 0x000000004 (3.5 rounded to 4)

fcvt.wu.s x12, f10 #expected result is 0xFFFFFFF0 (-3.5 rounded to -4)

fcvt.s.w f14, x11 #expected result is 0x40800000 (3.0 in float)

fcvt.s.wu f15, x12 #expected result is 0x40800000 (3.0 in float)

fmadd.s f16, f1, f2, f3 #expected result is 0x41400000 (12 in float)

fmsub.s f17, f1, f2, f3 #expected result is 0x40000000 (2 in float)

fmmadd.s f18, f1, f2, f3 #expected result is 0xc1400000 (-12 in float)

fmmsub.s f19, f1, f2, f3 #expected result is 0xc0000000 (-2 in float)

6.1.2. Logic elements and F_{max}

	Pipelined I	Pipelined F
Frequency F _{max} (MHz)	53.52 MHz	25.56 MHz
Total logic elements (%)	18,808 / 33,216 (57 %)	23,639 / 33,216 (71 %)
Total registers (%)	6,968 / 33,216 (21 %)	7,575 / 33,216 (23 %)
Total memory bits (%)	65,536 / 483,840 (14 %)	327,680 / 483,840 (68 %)

Table 18. Comparison Pipelined I and F.

6.2. Kit results

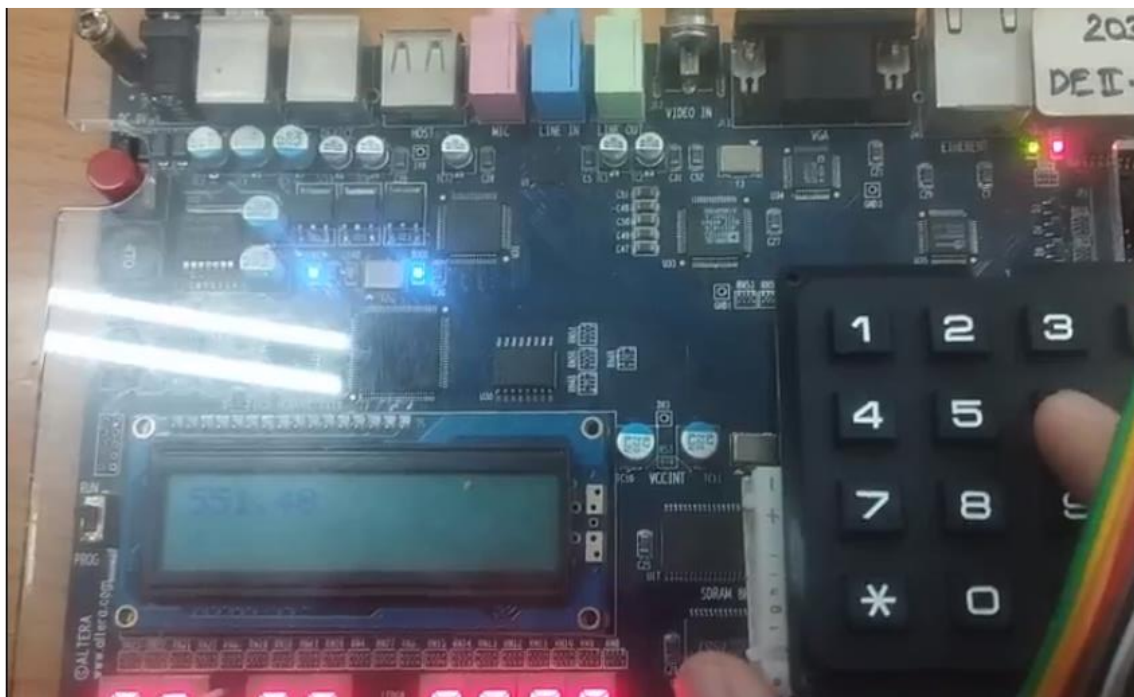


Figure 45. First Number and Operator Input.

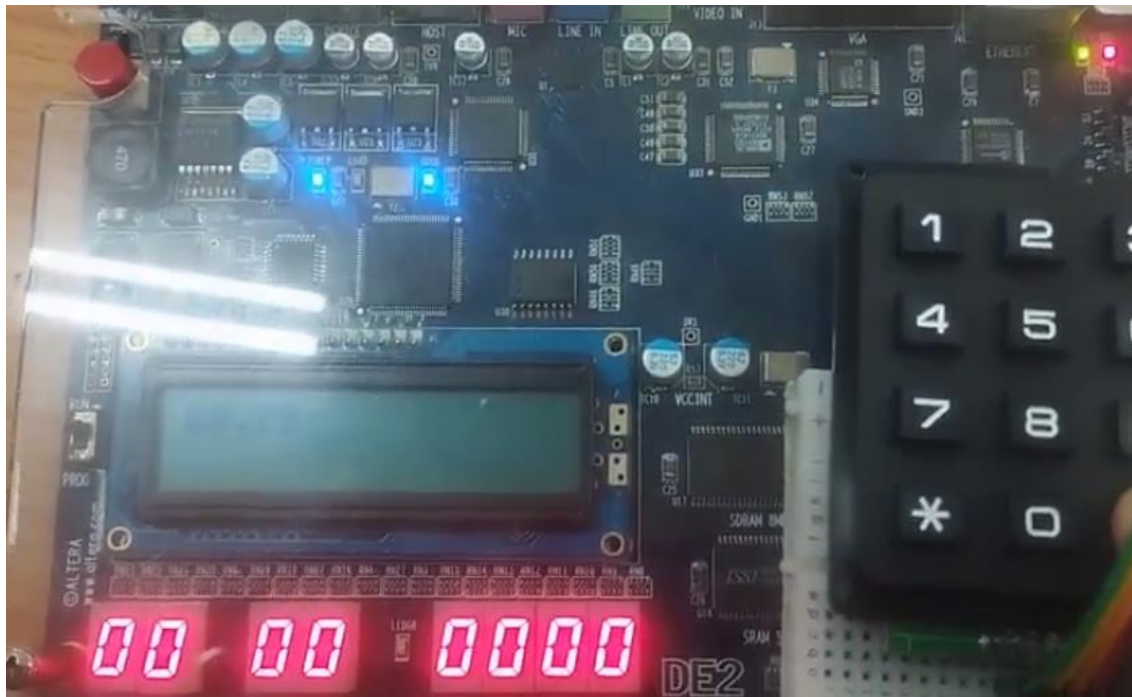


Figure 46. Second Number Input.



Figure 47. Stack Result.

7. Conclusion And Future Development

7.1. Conclusion

The implementation of a calculator using a RISC-V processor with the IF extension successfully demonstrates efficient floating-point arithmetic handling. By leveraging the Floating-Point Unit (FPU), the system ensures accurate and high-speed computations, while the structured pipeline minimizes execution delays. The integration of a keypad input system and display output enables seamless user interaction, making the calculator both functional and responsive.

7.2. Future Development

For future development, optimizations can be introduced to enhance performance and resource efficiency. Implementing hardware-accelerated division and further refining rounding mechanisms could improve computational accuracy. Additionally, expanding the calculator's capabilities to support trigonometric and logarithmic functions would make it more versatile. Exploring custom instruction extensions for floating-point operations in RISC-V could also enhance efficiency, paving the way for advanced embedded applications.

8. References

- [1] *ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*, IEEE Standard, New York, NY, USA, 1985.
- [2] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY, USA: Oxford University Press, 2010.
- [3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface [RISC-V Edition]*. Cambridge, MA, USA: Morgan Kaufmann, 2017.
- [4] K. K. Senthil Kumar, S. Yuvaraj, and R. Seshasayanan, "High-Performance Wallace Tree Multiplier," *International Journal of Computer Techniques*, vol. 7, no. 1, Feb. 2020.
- [5] P. Gogate and V. Saxena, "Investigation of a method to identify energy efficient organization of compression unit in parallel tree multipliers," *International Journal of Engineering Science and Technology*, vol. 10, no. 3, pp. 234–239, Mar. 2018.
- [6] S. Habakkuk and A. Akkas, "Design and Implementation of Reciprocal Unit Using Table Look-up and Newton-Raphson Iteration," in *Proceedings of the 7th EUROMICRO Conference on Digital System Design (DSD)*, 2004, pp. 249–253.
- [7] S. Mehrabi, R. F. Mirzaee, S. Zamanzadeh, and A. Jamalian, "A New Hybrid 16-Bit \times 16-Bit Multiplier Architecture by m:2 and m:3 Compressors," *International Journal of Information and Electronics Engineering*, vol. 6, no. 2, pp. 112–116, Mar. 2016.
- [8] T. Harish Anand, D. Vaithiyanathan, and R. Seshasayanan, "Optimized architecture for Floating Point computation Unit," in *2013 International Conference on Emerging Trends in VLSI, Embedded System, Nano Electronics and Telecommunication System (ICEVENT)*, Tiruvannamalai, India, 2013, pp. 1–5.
- [9] W. R. Dieter, A. Kaveti, and H. G. Dietz, "Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy," *IEEE Computer Architecture Letters*, vol. 6, no. 1, pp. 13–16, Jan. 2007, doi: 10.1109/L-CA.2007.4.
- [10] S. Javeed and S. S. Patil, "Low Power High Speed 24 Bit Floating Point Vedic Multiplier using Cadence".